2008

PERFORMANCE OPTIMIZATION OF A STRUCTURED CFD CODE - GHOST ON COMMODITY CLUSTER ARCHITECTURES

Pavan K. Kristipati
University of Kentucky, pavan.kristipati@gmail.com

ABSTRACT OF THESIS

PERFORMANCE OPTIMIZATION OF A STRUCTURED CFD CODE - GHOST ON COMMODITY CLUSTER ARCHITECTURES

This thesis focuses on optimizing the performance of an in-house, structured, 2D CFD code – GHOST, on commodity cluster architectures. The basic philosophy of the work is to optimize the cache usage of the code by implementing efficient coding techniques without changing the underlying numerical algorithm. Various optimization techniques that were implemented and the resulting changes in performance have been presented. Two techniques, external and internal blocking that were implemented earlier to tune the performance of this code have been reviewed. What follows is further tuning effort in order to circumvent the problems associated with using the blocking techniques. Later, to establish the universality of the optimization techniques, testing has been done on more complicated test case. All the techniques presented in this thesis have been tested on steady, laminar test cases. It has been proved that optimized versions of the code achieve better performances on variety of commodity cluster architectures chosen in this study.

KEYWORDS: Cache Optimization, Structured CFD Code Optimization, Efficient Coding Techniques, Improving Performance Without Changing Algorithm, Commodity Cluster Architectures.

Pavan K. Kristipati

12/3/2008

PERFORMANCE OPTIMIZATION OF A STRUCTURED CFD CODE - GHOST ON COMMODITY CLUSTER ARCHITECTURES

By

Pavan. K. Kristipati

Dr. Raymond .P. LeBeau
Director of Thesis

Dr. L. S. Stephens
Director of Graduate Studies

12/3/2008

RULES FOR THE USE OF THESIS

Unpublished thesis submitted for the Master's degree and deposited in the University of Kentucky Library are as a rule open for inspection, but are to be used only with due regard to the rights of the authors. Bibliographical references may be noted, but quotations or summaries of parts may be published only with the permission of the author, and with the usual scholarly acknowledgements.

Extensive copying or publication of the thesis in whole or in part also requires the consent of the Dean of the Graduate School of the University of Kentucky.

A library that borrows this thesis for use by its patrons is expected to secure the signature of each user.

Name Date

THESIS

Pavan K. Kristipati

The Graduate School

University of Kentucky

2008

PERFORMANCE OPTIMIZATION OF A STRUCTURED CFD CODE - GHOST ON COMMODITY CLUSTER ARCHITECTURES

THESIS

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in Mechanical Engineering
in the College of Engineering at the University of Kentucky

By

Pavan. K. Kristipati
Lexington, Kentucky

Director: Dr. Raymond .P. LeBeau
Assistant Professor of Mechanical Engineering
University of Lexington, Kentucky
2008

To my wife
Sudhira Kristipati

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and respect to my academic advisor Dr. Raymond P. LeBeau who has not only been a source of constant support and encouragement in many ways through out my Masters program but also who has given me an opportunity to complete my Masters program after being away from university for nearly 4 years. He is one of the very few people I came across who strives for excellence and creates an atmosphere for us to excel. Dr. LeBeau, I am honoured to be your student.

I would like to thank Dr. Johné M. Parker, Dr. T. Michael Seigler for accepting to be part of my thesis committee and for their valuable time in spite of their hectic schedule. Their input made this a better thesis. Sincere thanks to Dr. George Huang, for teaching how to understand a CFD code and for letting me work on his code.

A million thanks to my wife Sudhira, the backbone behind this accomplishment; it would not have been possible for me to wrap up this work after a gap of nearly 4 years without her help and her constant belief in me that I can do this. I want to thank my parents (Ashok and Visala) and Sudhira's parents (Prabhakara Rao and Bhanumathi) for believing in me. Dad – I finally made your life ambition come true!! My gratitude is beyond words for my uncle Satyam and aunt Bhavani, the couple who are the reason for me to be able to come to the USA and who constantly challenged me to work on completing my graduate degree.

Special thanks to the Graduate school at University of Kentucky for offering me the Kentucky Graduate Scholarship (KGS) and my life mentors Prakash Gupta and Dr. Santhosh Kumar.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF FILES

1. PKristipati-thesis.pdf 2-MB (file size)

CHAPTER – 1

1. INTRODUCTION

1.1 WHY CFD?

To face the demands from the market and the challenge from competitors, engineering firms consistently look for methods to reduce the time taken at any phase, whether it is a design process or a manufacturing process or a decision-making process. Computational Fluid Dynamics (CFD) is one particular science that many companies rely on in order to achieve time-compression. This is evident with the role CFD played in the recent development of Aston Martin racing car DBR9 [1]. Instead of the traditional route of using wind tunnels for design development, Aston Martin went straight from a CFD program to put the DBR9 on track. Another example of reliance on this science is a choice made by the BMW Sauber F1 team [2] to invest on performing simulations on its Intel® Xeon® dual- and quad-core processor-based supercomputer, rather than investing in a second wind tunnel. Heavy hydraulic equipment manufacturer Caterpillar effectively used computational fluid dynamics to make vital adjustments [3] to some of its hydraulic systems after making changes to a CFD model and analyzing the results. The various models allowed Caterpillar to improve the design of the tank. This prevented hydraulic pumps vehicles from failing before their expected shelf life as had been the case previously. While computational fluid dynamics has been in use at Boeing for since the mid 1970s, the most extensive application has been their newest commercial aircraft, the 787 Dreamliner [4]. The use of CFD tools has allowed Boeing to address a wide variety of design challenges, including traditional wing design, the even distribution of cabin air and a reduction in overall airplane noise. Also, CFD simulations shortened the development period of their latest aircraft *Dreamliner* by 18 months. Another example of benefit of application of this science has been in their wing development. In 1980, Boeing tested 77 wings in wind tunnels to arrive at the final configuration of their 767 model. 25 years later, they built and tested 11 wings for the 787, a reduction of over 80% in number of models tested. Those 11 wings required fewer resources (*viz.* man-power and time) and the wind tunnel results matched the CFD predictions. These are few examples of many instances in which CFD is being effectively used to reduce the time taken from design to manufacturing the product.

CFD has made it possible to predict many "what-ifs" under a given set of circumstances. Companies rely on this science because it is possible to predict how a particular design will perform before physical prototyping and testing. This reduces trial and error experimental testing processes thereby reducing the time it takes to manufacture a product. One classic example of this is in the aircraft industry. Before a prototype flies, the aerodynamics of an aircraft *viz.* lift, drag, side forces, moments must be determined. One option to obtain such aerodynamic data is to build many models and test them in wind tunnels under different test conditions. This is a costly process in regards to both time and money. Williams Grand Prix Engineering (Grove, England) [1] is an example of a company that with the help of computational fluid dynamics accelerated the development of their product Williams BMW FW27. CFD crash simulations on carbon fiber structures enabled this company to create energy-absorption plots that permit them to design parts without having to do lot of experimental crashes in order to find out the optimum balance between the energy absorption and weight reduction of the material.

1.2 SOLUTION PROCESS IN CFD

As presented in earlier section, although CFD is being widely used in real-world scenarios, understanding how to use it and being able to use it involves non-trivial processes that include problem analysis and access to computing power. This starts with understanding that fundamentally computational fluid dynamics deals with obtaining approximate computer-based solutions to a set of governing equations (conservation of mass, momentum and energy) that describe fluid flow. Obtaining analytical solution to these non-linear partial differential equations is not possible for most engineering problems and that is where computational fluid dynamics fills the gap.

The analysis process, shown in Figure 1-1, involves developing a computer model that performs CFD simulations. A typical approach is to first formulate the flow problem that is being simulated. The flow (computational) domain (control volume in which the flow field is computed) is then defined. Volume occupied by the fluid is then divided into discrete cells. This process is called grid generation. The next step involves specifying numerical conditions that are to be applied at the boundaries of the flow domain. These are called boundary conditions. Initial conditions of the flow field are then defined for

numerical methods to have a starting point. The fluid problem, defined by numerical equations, is then iteratively solved until the solution converges.

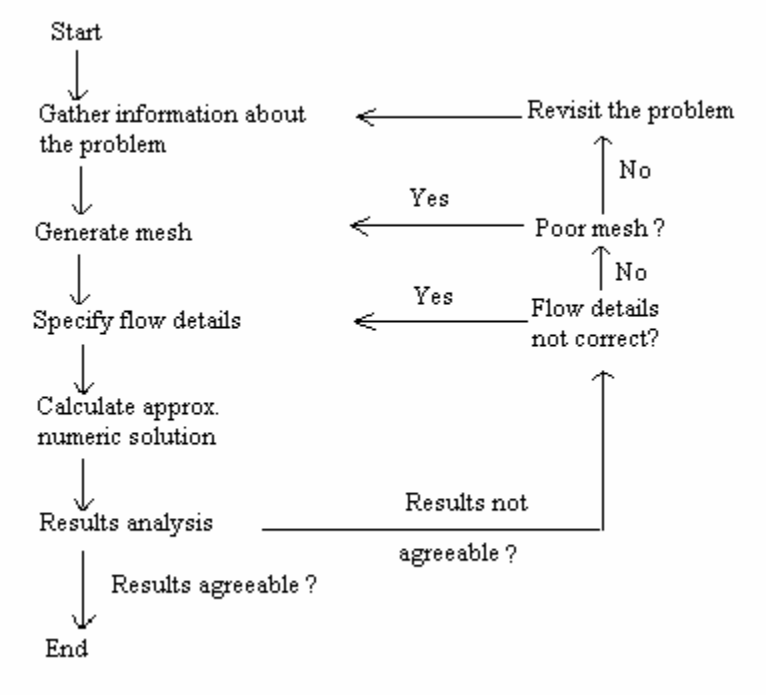


Figure 1-1 CFD analysis process

In CFD, a continuous problem domain is replaced by a discrete domain (a grid). In a continuous problem, flow variables like velocity, pressure, and temperature are defined at every point in the domain. For example, pressure p in a continuous 1D domain would be defined as:

$$p = p(x), \quad 0 \le x \le 1. \tag{1-1}$$

In a discrete domain, each flow variable is defined only at grid points. For example, pressure p in a discrete 1D domain would be defined at 'N' grid points as:

$$p_i = p(x_i), \quad i = 1,2,3,.....,N \tag{1-2}$$

The solution process involves solving for the relevant flow variables only at grid points. Values at other locations are calculated by interpolating values between grid points. The governing partial differential equations along with boundary conditions are defined in terms of continuous variables like p *and* V (velocity). In the discrete domain, these can be approximated in terms of discrete variables like p_i and V_i. This is illustrated with the help of a simple 1D example with *'N'* points on a grid as shown in Figure 1-2.

The discrete system involves a set of coupled, algebraic equations in discrete variables as discussed below with an example.

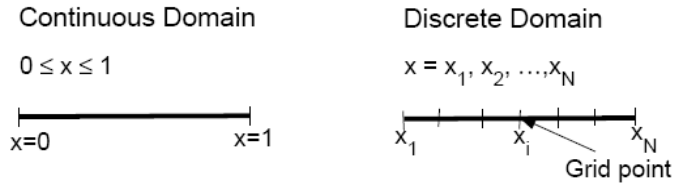


Figure 1-2 Continuous Domain and Discrete Domain

1.3 CFD AND COMPUTING POWER

In order to comprehend the amount of computing power needed to solve CFD problems, it is important to understand the fundamental ideas underlying CFD. To keep details simple, numerics behind the solution process are illustrated with a simple 1D equation shown below [76]:

$$\left(\frac{du}{dx}\right)_i + u^m = 0;\ 0 \le x \le 1;\ u(0) = 1\,. \tag{1-3}$$

When m=1 (for linear case), we have the equation:

$$\left(\frac{du}{dx}\right)_i + u_i = 0\,, \tag{1-4}$$

where i is any grid point. For simplicity, a 1-D grid with 4 points is considered as shown in Figure 1-3.

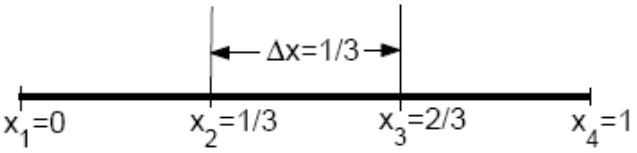


Figure 1-3 1-D grid

The grid has four equally-spaced grid points. The space between any two successive points is Δx . Taylor's series expansion in terms of u gives

$$u_{i-1} = u_i - \Delta x\left(\frac{du}{dx}\right)_i + O(\Delta x^2)\,. \tag{1-5}$$

Rearranging the above equation, we get

$$\left(\frac{du}{dx}\right)_i = \frac{u_i - u_{i-1}}{\Delta x} + O(\Delta x)\,. \tag{1-6}$$

Substituting Equation 1-4 in the above equation and neglecting the error, we get

$$\frac{u_i - u_{i-1}}{\Delta x} + u_i = 0 \ . \tag{1-7}$$

The above method of deriving a discrete equation from a differential equation using Taylor's series is termed as finite-difference method. Applying the above equation to 4 grid points on the grid and assuming u_1=1 as boundary condition, we get

$$u_1 = 1 \qquad (i=1), \tag{1-8}$$

$$u_1 + (1+\Delta x)u_2 = 0 \qquad (i=2), \tag{1-9}$$

$$u_2 + (1+\Delta x)u_3 = 0 \qquad (i=3), \tag{1-10}$$

$$u_2 + (1+\Delta x)u_4 = 0 \qquad (i=4). \tag{1-11}$$

The above system of equations comprises of four simultaneous algebraic equations. The above equations can be written in matrix form as

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ -1 & 1+\Delta x & 0 & 0 \\ 0 & -1 & 1+\Delta x & 0 \\ 0 & 0 & -1 & 1+\Delta x \end{bmatrix} \begin{bmatrix} u1 \\ u2 \\ u3 \\ u4 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Solving for u_1, u_2, u_3 *and* u_4 by inverting the matrix on the left hand side and using Δx=1/3, we get

$u_1 = 1, u_2 = 3/4, u_3 = 9/16, u_4 = 27/64$

The above example demonstrates the details of solving a simple 1D flow problem. In practical 2D/3D CFD applications, the above shown discrete system comprises of tens of thousands and possibly millions of equations. Setting up and solving such a large system involves an exceedingly high number of repetitive calculations. For example, the solution to a simple 2-D cavity flow problem (described in chapter 3) on a 600x600 (=360,000 grid points) grid essentially involves inverting a matrix of order 360,000x360,000. Although the matrix is sparse, the problem is magnified in that it is an iterative process and the process needs to run repeatedly until the solution converges or the optimum result is achieved.

For industry problems, CFD simulations are even more demanding in time and computing power as the number of grid points is on order of millions. For example, Baggett, *et al.* [5] calculated that number of grid points required for accurate Large Eddy

Simulation (LES) of a turbulent boundary layer scales as $N \sim Re^2$ where Re is Reynolds number and is of order 10 to 100 million in airplane simulations.

1.4 PARALLEL COMPUTING AND BEOWULF CLUSTERS

Over the years CFD simulations have expanded from the traditional applications of aerospace engineers and meteorologists to more diverse problems. Typically, complex CFD problems are solved at a national supercomputer center or similar facilities. Woodward *et al.* [7] and Anderson *et al.* [8] describe large scale simulations done on large parallel machines located at national laboratories. For many years, such computing resources could be afforded by only heavily-funded organizations and government laboratories, with limited access to outside organizations and academia. Alternatively, shared-memory supercomputers built by IBM or HP could be purchased by universities and large corporations; however, these machines are expensive with relatively high maintenance cost and unclear upgrade paths. One more alternative is for these organizations to pay for accessing super computing facilities; however, a queuing system that is typical in such cases controlled the timeline of projects based on CFD simulations. Thus, the high up-front cost of CFD analysis placed limitations on the size and complexity of the problem that could be solved by researchers and engineers whose applications could have benefited from the capabilities of CFD codes.

As the science of CFD advanced, so did the need for super fast computing sources. Hardware and resources on single processor workstations and serial computers became the bottleneck for performing complex simulations. In response, Thomas Sterling and Donald Becker [9] at NASA's Center of Excellence in Space Data and Information Sciences (CESDIS), in 1994, built a parallel computer from Common Off The Shelf (COTS) components. This resulted in an economical solution for high performance computing needs (Spector 2000). Their design had 16 486DX4 class workstations interconnected by a 10 base-T Ethernet network. Linux, a free UNIX clone was the operating system. Their creation, which they called "Boewulf" [9] was an instant success. This concept of building parallel computers from COTS components quickly spread through out NASA and the CFD research community. Advancements in computer technology and advantages like low cost, flexibility and access to latest technology fuelled Network of Workstations (NOWs) [10] and Beowulf models of COTS to make

substantial gains in the High Performance Computing (HPC) market. This trend has not seemed to slow down as clusters that operate on Linux are paving the way for multi-site supercomputing systems such as NSF's Distributed Terascale Facility (DTF) [11]. Computing systems like this support research such as climate, storm, and earthquake predictions. Clusters like this scale out and challenge traditional big supercomputers.

The principle of parallel computing that has been around for around 3 decades forms the basis of cluster computing. In parallel computing, a large problem is broken into discrete parts that can be solved concurrently. Each of these parts is further broken down into a series of instructions. These instructions are then executed simultaneously on different CPUs. These computing resources can be a single computer with multiple processors, many computers connected by a network or combination of both. It is also possible to take advantage of computing resources that are not local. For example, computers on a wide area network or even the ones on Internet can be used when local computing resources are scarce.

A commodity cluster [12] uses several such off-the-shelf PCs or customized PCs connected via Ethernet to solve problems that would otherwise needed to be handled by a supercomputer. Advances in clustering technology that redefined the price to performance curve made companies and organizations to embrace commodity clusters as their computing platforms. For example, PSA Peugeot Citroën [3] had been using proprietary Unix OS computer platforms for its CFD simulations, but in 2006 it decided to move to a more standardized 400 processor cluster of compute nodes using AMD Opteron processors inter-connected by a fast Myrinet [13] network in Linux. A combination of commercially available FLUENT software and AMD Linux cluster now enables PSA Peugeot Citroën engineers with a fluid flow modeling capability customized to the required level of performance and stability, yet less expensive to purchase and maintain than the previous approach. As shown by above examples, clusters not only increase computational resources multifold, but also eliminate the wait time that is typical in a super-computing environment. This is because clusters could possibly be dedicated for a specific department or to a particular project.

1.5 INTRODUCTION TO PROBLEM

The last two decades have witnessed a great increase in the amount of computational resources. Supercomputers in the current decade have reached more than 100TFLOPS (FLOPS – Floating Point Operations Per Second, is one of the ways to measure a computer's performance, especially in fields of scientific calculations that make heavy use of floating point calculations) while Cray machines of the early 1980s were operating at just a few Gigaflops/s. Such a rate of increase in peak performance shows no signs of slowing down. For example, the mythic Peta Flops (PFlops) has been achieved by IBM's Roadrunner in June 2008 [14] much earlier than a previous prediction of 2010 [15]. Such unprecedented growth in computational resources has become essential key for numerical simulation in industrial design and scientific research. But, such advancements in computer hardware technology is by no means a complete solution to the high computing needs for CFD in science and engineering. For example, Moin and Kim [16] report that even with a sustained performance of 1 Teraflops, even to simulate just one second of flight time of an airplane with 50-meter-long fuselage and wings with a chord length of 5 meters, cruising at 250 m/s at an altitude of 10,000 meters would take several thousand years. Spalart [6] estimated that even if computer performance continues to increase, Large Eddy Simulation (LES) for an aircraft will not be feasible until 2045 due to the high complexity of the problem.

Although the challenges caused by lack of access to suitable computing resources or lack of large amounts of money associated with accessing or owning such facilities could be overcome with cluster computing technology, new challenges started to arise. Coordinating concurrent operation of many processors (hundreds if not thousands), which is the basis of cluster computing, is a complex task that requires sophisticated software related tools *viz.* parallelized versions of CFD codes, debuggers, analysis tools and communication libraries. Because of this, the search for new and efficient algorithms and computational techniques became heart of CFD and so did the role of numerical mathematics.

The search for efficient algorithms is also fuelled by non-uniform growth in computer hardware. For example, the processing speed of a CPU has historically increased at a rate of about 55% per year, whereas the main memory access speed has

increased at a rate of only 7% per year [17]. This means over the years the gap in the speed to access the memory increased and this has been estimated at 45% per year [18]. Because of this performance mismatch, processors rely on caches (discussed in detail in chapter 2) to reduce effective memory access time. Even with the introduction of caches and other advancements in hardware, present-day compilers can perform certain simple code optimizations, but they are not sophisticated enough to change the codes so that they can make best use of the memory hierarchy. Meanwhile, scientific programs tend to be particularly memory-intensive and dependent on memory hierarchy. The key to achieving high performance is an optimal architecture-algorithm mapping. It is not uncommon to experience poor performance when such mapping is not established. For example, in 1992, a simulation study by Mowry *et al.* [18] discovered that scientific programs spend from a quarter to half of overall execution time waiting for data to be fetched from memory during sequential execution. It was noticed by Beyls et al [19] that the processor stalled on data memory access for almost 50% of the execution time for the SPEC2000 programs which were compiled with the highest level of optimization present in Intel's state-of-the-art compiler. Thus, achieving high performance on modern architectures is intimately related to the coding style. This is particularly true with respect to CFD codes as almost all of them are numerically intensive. Otherwise, while the peak performance of workstations and parallel machines increases, the gap between peak and actual performance of codes becomes wider when no attention is paid to optimal memory utilization. As it would be observed in later chapters, this effort to make memory utilization optimal has resulted in dramatic performance gains in a CFD code.

1.6 PRESENT WORK

Over the years, although there have been positive claims in favor of commodity clusters, achieving optimum code performance on them involves non-trivial effort *viz.* carefully engineering the cluster design, using tools to improve application performance and restructuring the code. The present work deals with optimizing performance of a CFD code, GHOST, on commodity clusters. These clusters are essentially modern cache-based processor architectures. Although CFD codes are now rarely run on a single node, the present work first focuses on tuning the code on a single node. This is because performance of a parallel code is a combination of both single node performance and

code scalability across an increasing number of nodes. Improvements in one of these two areas may be masked by lower performance in the other, complicating the already non-trivial optimization process. Also, code performance can vary unexpectedly with changes in grid size as circumstantial choices can lead to fortuitous or detrimental memory storage and cache performance. As discussed in the previous section, the fundamental idea of the present work is to achieve optimum memory usage by mapping the algorithm to the memory architecture without modifying the underlying algorithm. Later, the fine-tuned code on a single processor is run on multiple nodes and scalability of the code is presented. Subsequently, lessons learned on a commodity cluster have been applied on other platforms; similar or better performance gains have been observed as explained in further chapters.

The present work presents the results of optimization effort on a two-dimensional CFD code GHOST on commodity cluster architectures. Some of the techniques that have been applied to improve performance of the code are presented in chapter 2. The GHOST code is discussed in great detail in chapter 3. This code is extensively used across several commodity cluster platforms KFC3, KFC4, KFC5 and KFC6. Details of these clusters along with the test case are discussed in chapter 3. The goal of the present work is to minimize the walltime (*viz*. the amount of time that passes if you are looking at a clock on the wall for the code to finish solving a problem) of the code while maintaining the accuracy of the code and without altering the solution from the code. Another aspect of the work is to apply the lessons learned in the optimization process and apply them to different architectures. Essentially, there are two stages in this tuning effort. First stage of tuning effort was focused on tuning GHOST on KFC3 and KFC4; it was carried out till December 2004. The second stage (September – November 2008) of tuning effort comprises of testing the tuned codes on KFC6 architectures. These results are presented in chapter 4 along with the results on KFC3 and KFC4 for comparison purposes. External and Internal Blocking techniques that were used to tune GHOST in the interim are reviewed in chapter 5. Later, results of subsequent tuning effort (that is part of second stage of the tuning effort) are presented in chapter 5 along with presenting the results on a second test case. Conclusions and future work are presented in chapter 6.

CHAPTER – 2

2. CACHE-BASED ARCHITECTURES

2.1 INTRODUCTION

As a programmer, it would be ideal to not to need to know about the details of an underlying computer architecture. However, there have been dramatic changes in the design of computer architectures in the past two decades. Memory chips and micro-processors have increased in performance exponentially over time [20]. Cache has been introduced in computer architectures to bridge the gap between the processor speed and memory speed. When a scientific programmer understands these developments and exploits the knowledge of the memory hierarchy, it is possible to achieve impressive speedups without modifying the underlying algorithms. This is particularly observed in CFD codes which generally apply mathematically simple but repetitive operations to a large set of data. In such codes, the number of times the data is piped in and out of the CPU from the memory is often a limiting factor for performance. This correlation is discussed in detail in this chapter. Also, in the case of computers with distributed memory [21], the speed of interprocess communication also plays a major role in the overall performance of a code, with slower speeds of accessing data on another processor further limiting the code's speed. Thus details of the computer architecture have a significant impact on the speed of a code running on a given machine.

2.2 EVOLUTION OF CACHE-BASED ARCHITECTURES

Early designs of Personal Computers (PCs) had processors running at ~8 MHz or less. It was not often that the processor would be waiting for the system memory. It did not matter much if the memory was slow, the processor was not fast either. Within a few years of the invention of the PC, every component had increased in speed. However, some increased far faster than others. Memory and memory subsystems are now much faster than they were, by a factor of 10 or more. However a current top of the line processor has a performance over 1000 times that of the original IBM PC (4.77 MHz) [22, 23]. This disparity in performance improvement has left us with processors that run much faster than everything else in the computer. These powerful processors would not be giving their best performance without the use of special high-bandwidth memory

reserves called cache. Without cache, getting data and instructions would be bottlenecked by the relatively snail-paced capability of the system memory or Random Access Memory (RAM). This is the reason why modern microprocessors have cache. Almost all of them have multiple layers of it. As cache memory is relatively expensive (a 4 MB of Compaq L2 cache costs ~ $400 [24]), instead of trying to make the whole of systems memory (RAM) faster, a smaller piece, typically starting with few KB, is often a starting point. Cache is then used to hold information most recently used by the processor, as in general the processor is more likely to need information it has recently used, compared to a random piece of information in memory. Details about the internal working of cache memory are presented in later sections.

During 1961-62, a research group at Manchester [25, 26], England introduced the concept of *virtual memory*. This gave the programmer the illusion that he had access to an extremely large main memory even though the computer actually had a relatively small main memory [27]. They came up with an algorithm that would move the information that was not currently being used, back into the secondary memory viz. hard drive. All this was carried out by the operating system. This concept was widely used in most of the operating systems in the 1960s. In 1965 Maurice Wilkes [28] proposed the "slave memory", which was a small fast access storage device on the processor to hold a small amount of the instructions and data most recently used by the processor. This was later called "Cache Memory" in 1968 when IBM introduced it on the 360 / 85 machines [29]. Cache memory is now a standard part of memory architecture.

2.3 MEMORY ARCHITECTURE

While a cache-based CPU is a common design, the specifics of the hardware structure vary from processor to processor. In order to be able to maximize code performance on commodity clusters, it is essential to maximize the code performance on a single processor. To do this, it is important to understand the underlying memory architecture. The following section presents a brief discussion of memory architecture.

Most modern day computer systems have multiple levels of memory as shown in the Figure 2-1. Each level is of a different size and operates at different speed. The fastest is the closest to the CPU and each subsequent layer gets slower, farther from the processor.

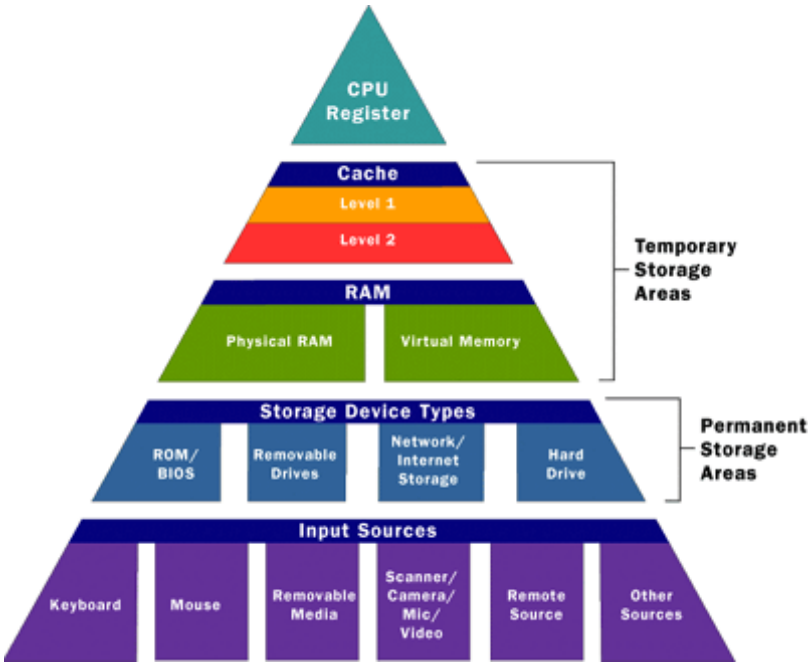


Figure 2-1 Memory hierarchy in a modern day computer [30]

At the top of memory hierarchy is the Central Processing Unit's (CPU) general purpose registers. Registers provide the fastest access to data (less than 1 clock cycle) and are the smallest memory object in the memory hierarchy. These are also the most expensive memory locations. Processors can only work on the data available in the register. Registers are measured by the number of bits they can hold viz. an "8-bit register" or a "32-bit register".

The next highest performance subsystem in the memory hierarchy is the Level one (L1) cache [31]. Although L1 cache size is quite small (4 KB to ~256 KB), its size is much larger than the registers on the CPU. Most memory architectures have Level two (L2) cache as part of the CPU package. L2 cache is generally larger (256 KB to few MB) than the L1 cache and is a secondary staging area that feeds the L1 cache. L2 may be built into the CPU chip, reside on a separate chip, or be a separate bank of chips on the motherboard. Generally, L1 and L2 caches are split into instruction and data caches. If the data present in the L1 cache is also present in the L2 cache, they are called inclusive (e.g. Intel Pentium 2, 3 and 4). If the data is present at most in either the L1 or the L2 cache, they are called exclusive (e.g., AMD Athlon). Exclusive caches can hold more data compared to inclusive ones, the downside being the penalty incurred while

transferring the data from L2 to L1 cache. In inclusive caches, data from L2 is directly written on L1 by deleting some part of the data already present.

L1 and L2 caches are made out of expensive SRAM (Static RAM) memory. SRAM is distinctly different from the main memory viz. DRAM (Dynamic RAM). A few differences are outlined in Table 2-1. SRAM uses more transistors for each bit of information; it draws more power and takes up more space for this reason.

Table 2-1 Differences between SRAM and DRAM

SRAM	**DRAM**
Static RAM	Dynamic RAM
Faster and expensive than DRAM	Slower and inexpensive than SRAM
Access times of ~10 nanoseconds	Access times of ~60 seconds
Does not need to be refreshed like DRAM	Needs constant refresh
Generally used for cache (viz. L1, L2)	Generally used for system memory

System Memory (RAM) [30] is another kind of data storage used in a computer and is present between the cache and the hard drive. It can be thought of as a larger and slower cache which allows random access to the data that is stored on it. Similar to the cache, RAM loses its data when the computer is switched off. It takes the form of integrated circuits (ICs) that allow the data to be accessed in any order, i.e., at *random*. The word *random* thus refers to the fact that any piece of data can be returned to the requestor in the same time regardless of its physical location and whether or not it is related to the previous piece of data.

2.4 CACHE'S ROLE IN THE PERFORMANCE OF A CODE

When compared to the size of a hard disk, the size of cache is usually small. Yet, its effective usage helps in increasing the speed of the program execution. This section presents cache's role in the performance of a code.

2.4.1 LOCALITY OF REFERENCE

The Principle of *Locality of Reference* states "Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program

spends around 90% of its execution time in only about 10% of the code." [20]. Locality can be subdivided into temporal locality and spatial locality.

Temporal locality – A sequence of references exhibits temporal locality if recently accessed data/instructions are likely to be accessed again in the near future.

Spatial Locality – A sequence of references exhibits spatial locality if data located close together in address space tend to be referenced close together in time.

Hence if a code's algorithm could supply this information (that is used 90% of the time) readily to the processor, performance improvements can be achieved. This is where cache plays an important role in performance of a code. Once the data is stored in the cache, future use can be made by accessing the cached copy rather than re-fetching or re-computing the original data. This reduces the average access time to this piece of data since the access time increases as we move further away from the registers towards RAM as shown in Table 2-2. In order to demonstrate why *locality of reference* works, a pseudo-code is presented in Figure 2-2.

Table 2-2 Characteristics of memory types

Type	Typical Access Speed	Latency	Size
Registers	~2 nanoseconds	~ 0 - Cycles	~1b
L1 Cache	~10 nanoseconds	~1 – Cycle	~ 4 KB – 256 KB
L2 Cache	~20 –30 nanoseconds	~ 10 – Cycles	~ 128KB – 4 MB
RAM	~60 nanoseconds	~ 100 – Cycles	~ 128 MB – 4GB
Hard Disk	~ 10 milliseconds	-	~ 20GB – 500 GB

This program asks the user to enter a number between 1 and 1000. It reads the value entered by the user. Then, the program divides every number between 1 and 1000 by the number entered by the user. It checks if the remainder is zero (integer division). If so, the program outputs "C is a multiple of A". Then the program ends. Out of the 10 lines of this program, the *loop* part (lines 6 to 9) of the program is executed 1000 times. The remaining lines are executed only once. Lines 6 to 9 will run significantly faster because of caching. As this program is very small, it can entirely fit in the cache memory.

```
print "Enter a number between 1 and 1000"
read (a)
c=a+1
for (i=0;i<1000;i++)
{
 d=c%a (remainder - integer division)
 if (d==0)
 print "c multiple of a"
 c=c+1
}
```

Figure 2-2 Illustration of working of locality of reference

Even in larger programs, a lot of processing happens inside loops. For example, a word processor spends 95% of the time waiting for user input and to display it on the screen [31]. This part of the word-processor program is put in the cache. In case of a word processor, this 95% to 5% is what is called as *locality of reference.* Locality of reference can be exploited when an algorithm makes best use of cache memory because caches are faster memory subsystems specially designed to store recently referenced data and data near recently referenced data. This can lead to potential performance increases.

2.4.2 CACHE HIT AND CACHE MISS

During a computation, if the processor requests data, it is first searched for in the L1 cache. If it is found in the L1 cache, it is called a L1 cache hit; otherwise it is called a L1 cache miss. Then the data is searched in the immediate lower (in hierarchy) memory, in this case the L2 cache. If the data is found in the L2 cache it is a L2 cache hit or if the data is not in the L2 cache the next higher memory is searched and it is called a L2 cache miss. This process continues with however many levels of cache the system has before the processor has no other option than to retrieve the data from external memory, the slowest option of them all. Assuming that the requested data is found in main memory, it is copied from main memory along with neighboring bytes in the form of cache block or cache line, into the L2 cache and then into L1 cache. When the CPU requests this data again, if this data is found in the L1 cache, it is a L1 cache hit or else a L1 cache miss and then the above described process repeats again till data is found. Hennessy and Patterson

[34] classified cache misses into three categories depending on the situation that brought about the cache miss:

Compulsory cache misses is a cache miss that occurs because the desired data was never in the cache and therefore must be brought in for the first time during a program's execution. These are also called *cold start misses* or *first reference misses.*

Capacity misses are those misses that occur due to the fact that a particular data block was moved out of cache to accommodate other blocks of data. This is due to the fact that the cache memory is of finite size and so cannot accommodate all blocks that are needed for program's execution.

Conflict cache misses are those misses that occur because an earlier entry was evicted. This type of misses can be further broken down into *mapping* misses, that are unavoidable given a particular amount of associativity, and *replacement* misses, which are due to the particular victim choice of the replacement policy. *Conflict* cache misses are discussed in detail in later sections.

As a cache miss refers to a failed attempt to read or write a piece of data in the cache, cache misses can also be classified based on if it is an instruction miss or data miss.

A cache *read miss* from an *instruction cache* causes the most delay, because the processor has to wait until the instruction is fetched from memory. However, instruction cache misses do not have significant impact on performance of numerically intensive codes *viz.* CFD codes as most of their execution time is spent in small computational kernels based on loop nests though repetitive do not involve complex calculations.

A cache *read miss* from a *data cache* happens when a particular piece of data requested by the CPU is not found in the cache. This type of cache misses has the most impact on the performance of a numerically intensive code.

A cache *write miss* to a *data cache* generally causes the least delay because the write can be queued. The processor can continue with its operation until the queue is full. But, in numerically intensive codes, this might not always be true as the code might require this updated data in subsequent instructions and the processor might have to wait till the value is written to the RAM.

So ideally, for the fastest execution of any code, we need to have data stored in such a fashion that there are no cache misses. This is possible only for small grids which fit into L1 or L2 cache, but this may be impractical for large CFD calculations. So the idea is to reduce the L1 and L2 caches misses and bring them as close as reasonable to zero. A few techniques to achieve this are described in detail in later sections of this chapter.

From Table 2-2, it is evident that cache misses lead to a reduction in the efficiency of the code due to increased delay in data or instruction access. When the processor is unable to find the necessary data in the cache, it has to go look for it in the main memory or random access memory (RAM). This leads to a latency of around 60 nanoseconds. Over the years, this latency difference between main memory and the fastest cache has become larger. For example, Clark *et. al* [35] in 1983, report that the time to service a cache miss to memory for the Vax 11/780 machine was 6 cycles while Fenwick *et. al* [36] in 1995, report that it is 120 cycles for AlphaServer 8400. Because of this, some processors have begun to utilize three levels of on-chip cache. For example, in 2003, Itanium2 began shipping with a 6 MiB (1 Mebibyte (MiB) = 2^20 bytes ~ 1 MB) unified Level 3 (L3) cache on chip [35]. The IBM Power 4 series has a 256 MiB L3 cache off chip, shared among several processors. The new AMD Phenom series of chips carries a 2MB on die L3 cache.

Presence of multiple levels of cache does not automatically mean better cache-hit rate or better performance. Cache-hit rate often correlates to the program's locality of reference, meaning the degree to which a program's memory accesses are limited to a relatively small number of addresses. Conversely, a program that accesses a large amount of data from scattered addresses is less likely to use cache efficiently.

2.4.3 HOW CACHE MEMORY WORKS

In order to have a deeper understanding of cache memory's role in the performance of a code, it is important to understand the internal working of cache memory. When a computer is turned on, cache memory is empty and so for first instruction/data request, access to RAM is compulsory. Since usually programs flow in a sequential manner, the next memory position the CPU will request is probably be the position immediately below the memory position that got loaded. After the first

instruction/data is loaded from a certain memory position, a circuit called the memory cache controller loads a small block of data below the current position that the CPU has just loaded. This amount of data is called a *cache line* and is usually 16 to 128 bytes long [31]. Typically, it is 64 bytes for most caches. This prevents the CPU from reaching to RAM for data access. Besides loading this small amount of data, the memory controller always tries to guess for what the processor will ask next. A circuit called the *prefetcher* loads more data located after these first 64 bytes from RAM into the cache memory. If the program continues to ask for instructions and data from memory positions in a sequential manner, they are already available in the cache memory because of caching. This process can be summarized in a few steps as shown below:

- CPU asks for instruction/data stored in address 'x'
- Since the contents from address 'x' are not inside the cache memory, this will be fetched from RAM.
- Cache controller loads a line (typically 64 bytes) starting at address 'a' into the memory cache. This is more data than the CPU requested; so, if the program continues to run sequentially, (i.e., asks for address x+1) the next instruction/data for which the CPU will ask is already in the cache.
- A circuit called prefetcher loads more data located after this line, i.e., starts loading the contents from address x+64 into the cache. For example, Pentium 4 processors have a 256-byte prefetcher, so it loads the next 256 bytes after the line already loaded into the cache.

However, programs do not run in a sequential manner always. The control jumps from one memory location to the other, some times at random. The main challenge of a cache controller is to guess what address the CPU will ask in future so that this address can be loaded into cache to avoid CPU from going to RAM to fetch this address. This task is called *branch predicting* [30] and all modern CPUs have this feature.

2.5 CACHE MEMORY ORGANIZATION

Internally, cache memory is divided into lines, each line holding from 16 to 128 bytes, depending on the CPU. What follows is a discussion of how memory cache is organized using 64-byte lines as an example. Figure 2-3 presents how a 512 KB L2 cache memory is divided into 8192 lines (512 * 1024 / 64 = 8192).

Cache memory can be classified into 3 types based on how it is mapped with RAM.

1) Direct mapping
2) Fully associative
3) Set associative (also called n-way set associative)

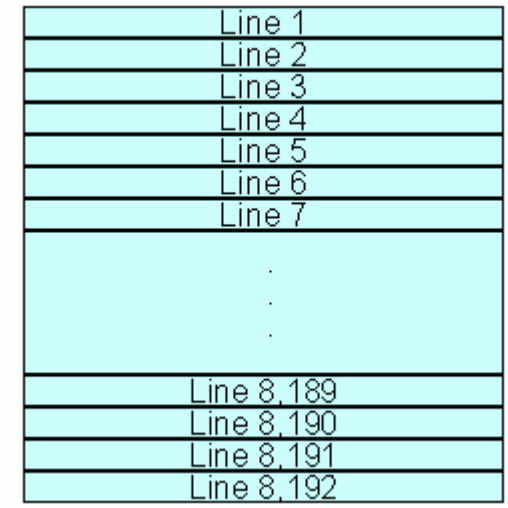


Figure 2-3 512 KB L2 memory [31]

Direct Mapping: In this configuration, RAM is divided into the same number of lines as the cache memory. For example, a 1 GB RAM will be divided into 8192 blocks (assuming cache memory uses the configuration shown in Figure 2-4) and so each block is 128 KB. This is illustrated in Figure 2-4.

The main advantage of direct mapping is that it is the easiest configuration to implement. When the CPU asks for an address from RAM (example address 2000), the cache controller loads a cache line (64 bytes) from RAM into cache memory. These addresses (from 2000 to 2063) are stored in cache. If CPU requests any of these addresses, they are already available in cache.

The problem surfaces when CPU requests two addresses that are mapped to the same cache line. Since there is only one possible place that any memory location can be cached, there is nothing to search. The cache line either contains the memory information being looked for, or it does not. For example, assume CPU requests two different addresses *A* and *B* that map to the same cache line, in alternating sequence (*A, B, A, B*). This could happen in a small loop. The processor will load *A* from memory and store it in cache.

Then it will look for *B*, but *B* uses the same cache line as *A*, so it would not be available. So, *B* is loaded from memory and stored in cache for future use. But, then the processor requests *A* and looks for in the cache. It finds *B* instead of *A*.

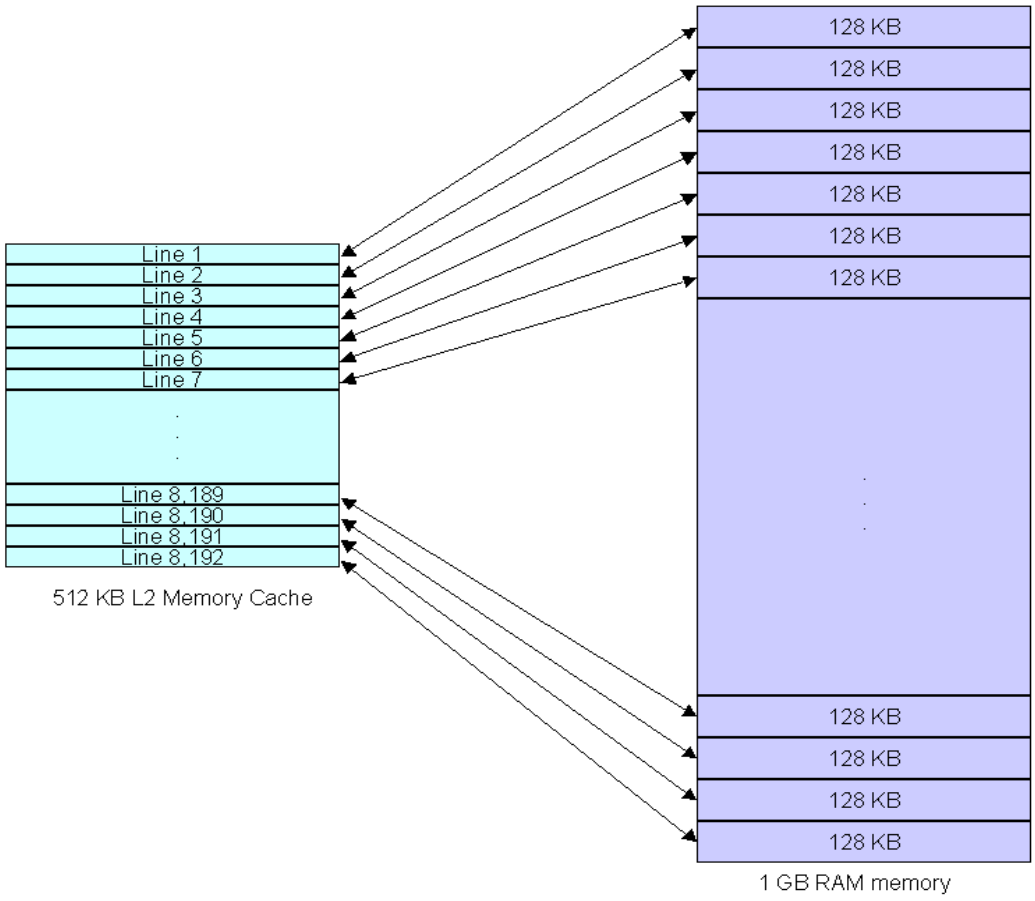


Figure 2-4 Direct Mapping cache [31]

This conflict occurs repeatedly. These are called *Collision* or *Conflict* misses [34]. The net result is that the hit ratio, in this example, is 0%. This is a worst case scenario. But in general, the performance for this type of mapping is worse compared to the other two mappings.

Also, if the program has a loop that is more than 64 bytes (cache line) long, cache misses are experienced for the entire duration of the loop. For example, if the loop goes from address 1000 to address 1100, the processor will have to load all the instructions/data from RAM as long as the program's control is inside the loop (which is 90% of the time as depicted in example at the beginning of this chapter, especially in numerically intensive codes). If the loop is executed 1000 times, the processor will have

to go to RAM to fetch the data/instructions leading to adverse impact on code's performance. This is why direct mapping is considered to be the least efficient cache configuration.

Fully Associative: In a fully associative configuration, there is no hard linking between memory addresses in RAM and cache lines. The cache controller can store any address. Thus, the problems that surface in direct mapping configuration do not occur in this case. Although this makes this configuration the most efficient, the control circuit is far more complex as it needs to keep track of what memory locations are loaded inside the cache memory. To mitigate the disadvantages of direct mapping and to take advantage of fully associative cache mapping, a hybrid solution called *Set Associative* is used most often.

N-way Set Associative: In this configuration, cache memory is divided into several blocks (sets), each block containing 'n' lines. For example, a 4-way associative cache (of 8192 cache lines) contains 2048 blocks having 4 lines each. This is shown in Figure 2-5. In this type of cache configuration, the system's RAM is divided into the same number of blocks as the cache memory. Thus, in our example, 1 GB RAM is divided into 2048 blocks each of 256 KB as shown in Figure 2-6.

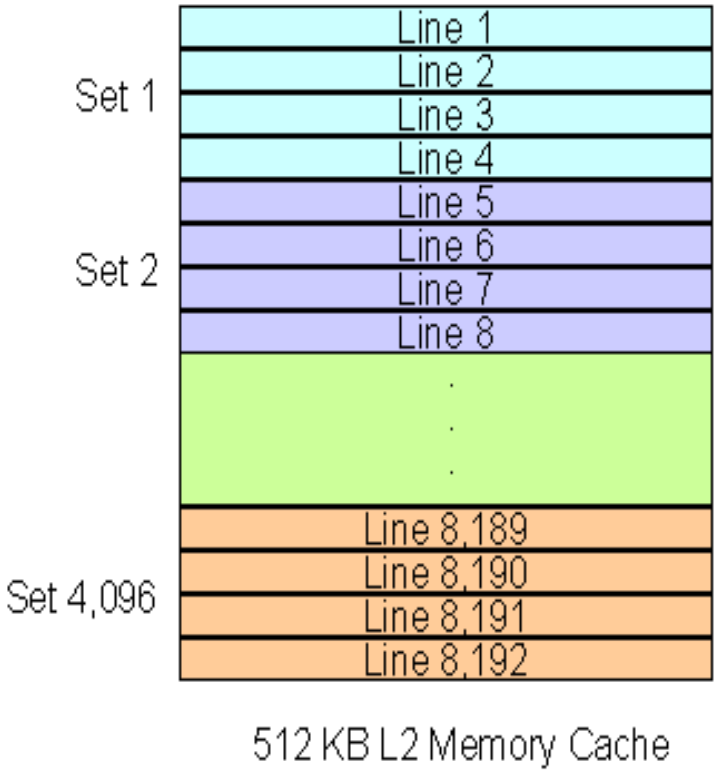


Figure 2-5 4-way associative 512 KB L2 cache memory [31]

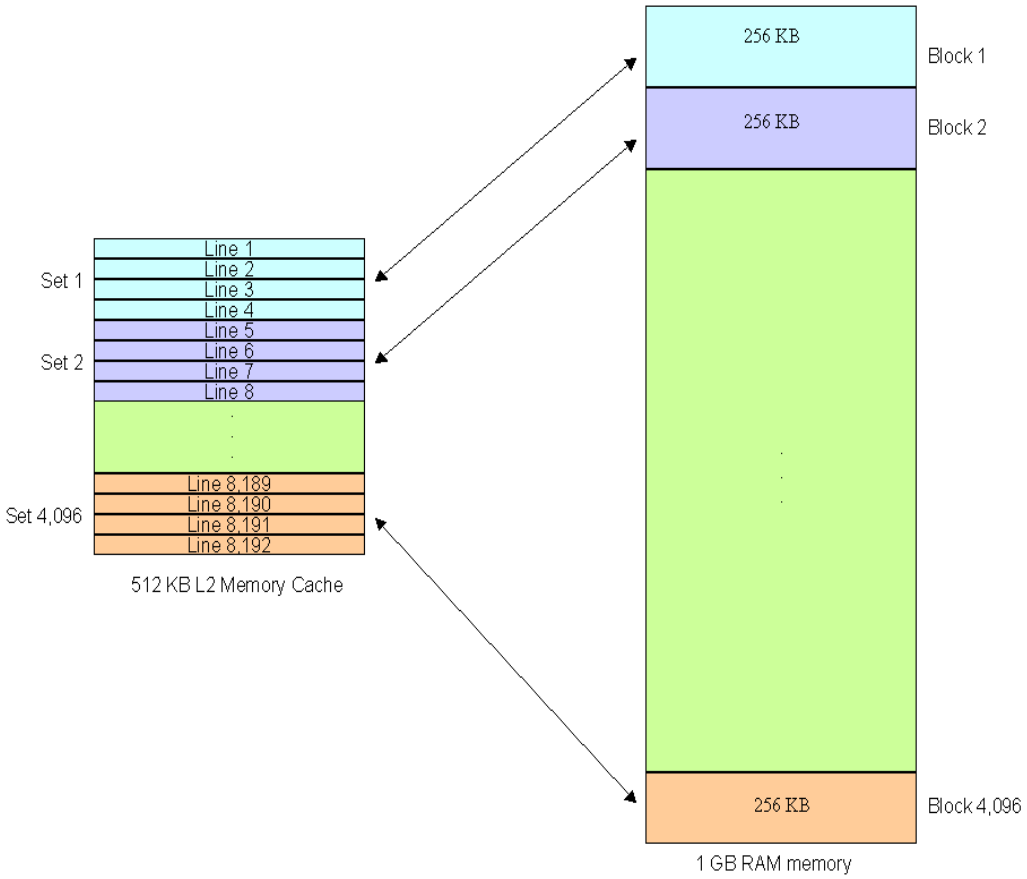


Figure 2-6 512 KB L2 cache memory configured as 4-way associative [31]

As can be observed, mapping in this case is very similar to direct mapping; the difference being for each memory block, there is now more than one cache line available on cache memory. Each cache line, in this type of configuration, can hold the contents from any address inside the mapped block on RAM. For example, on a 4-way set associative cache, each memory address inside a mapped block is assigned a set, and can be cached in any one of 4 locations within the set that it is assigned to. In other words, within each set the cache is associative, and thus the name. With this design, the problems (*viz.* collision, conflict and loop) presented by direct mapped cache are mitigated. Added to this, because of the ease of implementation, this type of cache configuration is the most used in PCs, although it provides lower performance compared to the fully associative cache configuration.

This design means that there are "*N*" possible places that a given memory location may be in the cache. The trade off is that there are "*N*" times as many memory locations competing for the same "N" lines in the set. In the example discussed above, instead of a single block of 8192 lines, we have 4096 sets with 4 lines in each set. Each of these sets is shared by 4096 blocks of memory on RAM each of 256 KB size. As each set has 4 cache lines in it, each address can be cached in any of 4 cache lines. This means that in the example described in the direct mapped cache description above, where two addresses (*A* and *B*) that map to the same cache line were accesses alternately, they would now map to the same cache set instead. This set has 4 lines (in 4-way set associative) and so one could hold *A* and the other could hold *Y*. This raises the hit ratio from 0% to 100%. Table 2-3 summarizes different cache mapping techniques and their relative performance.

Table 2-3 Mapping techniques and their relative performance [31]

Cache Type	Hit Ratio	Search Speed
Direct Mapped	Good	**Best**
Fully Associative	**Best**	Moderate
N-Way Set Associative, N>1	Very Good, Better as N Increases	Good, Worse as N Increases

The mapping between memory block and cache lines (which memory block goes into which cache line) and replacing the contents of cache line is decided by a replacement strategy. The most commonly used strategies for today's microprocessor caches are random and least recently used (LRU). The random replacement strategy chooses a random cache line to be replaced. The LRU strategy replaces the block which has not been accessed for the longest time interval. This confirms with the principle of locality in that it is more likely that a set of data that has been recently used would be used in the near future. Less common strategies are least frequently used (LFU) and first in, first out (FIFO). The former replaces the memory block in the cache line which has been least frequently used, whereas the latter replaces the data that has been residing in

cache for the longest time. Eventually, the optimal replacement strategy replaces the memory block that has not been accessed for the longest time. It is not practical to implement such an *optimal* replacement strategy in real world scenarios as such a design would require information about future cache references.

2.6 CACHE OPTIMIZATION GUIDELINES

Although the market that caters to the scientific community is diverse, there is still a certain convergence in the architectural design of machines. For example, RISC (Reduced Instruction Set Computer) architectures from many vendors (IBM, SUN etc) are relatively similar. Registers, cache, memory, and disk are now present in one form or another in all architectures of practical interest. Consequently, understanding how to achieve high performance on a given architecture is often of sufficient generality to allow efficient computations on architecturally similar computers. As machines evolve, new and improved numerical algorithms need to be developed that not only solve the equations but also take fuller advantage of the advances in the architecture of the computers on which they run. The key to achieving high performance is an optimal architecture-algorithm mapping. Since the effective use of cache memory is critically important to any overall code performance, numerous research papers have discussed cache optimizations in the last forty years [38]. The proposed optimizations range from hardware modifications, over micro architectural enhancements, optimizations in compilers and operating systems, to improvements at the algorithmic level. This section presents cache optimization methodologies for better code performance.

Optimizations techniques presented below can be classified into the following categories:

- Optimizing memory access
- Optimizing floating point calculations
- Using compiler optimizations

Each of these will be discussed in subsequent sections.

2.6.1 TECHNIQUES FOR OPTIMIZING MEMORY ACCESS

The techniques that optimize memory access also reduce capacity misses. They essentially aid in holding on the data in cache for a longer time so that as many necessary

calculations as possible in which the data is required are carried out before this data leaves the cache. Some of these are presented below.

2.6.1.1 Optimal Data Layout

The idea is to ensure that the data processed in sequence should be located close to each other in physical memory. This ensures *data locality*, meaning that data that are brought in cache will be used at least once before being flushed out of cache. For example, a Fortran 77 array containing the positions of a collection of n particles should be dimensioned as dimension r(5,n)instead of dimension r(n,5)since typically the five spatial coordinates for a given particle are accessed consecutively. The same considerations apply to data structures defined in Fortran90. The effect of other performance improvement techniques on numerically intensive codes might be mitigated with a poor data layout as described above.

2.6.1.2 Loop Interchange

This technique suggests reversing the order of two adjacent loops, if needed, in a nested loop [40, 41]. The idea is to optimize the inner-loop memory access. In FORTRAN, the data is accessed row by row instead of column by column as in C and C++. As the value of the inner loop changes most frequently, this type of loop interchange results in a performance gain because the order of data access is similar to the order of data storage. It is critical to understand the order in which the data is accessed in the language in which the code is being written and design the order of nested loops to match with the data access strides.

In the untuned version of the code shown in Figure 2-7, the loop accesses the arrays *x, y* and *z* row by row while FORTRAN program stores array elements in *column-major* fashion. (Elements from the same column example: x[1,1], x[2,1], x[3,1] are stored together). This might result in heavy cache misses as the contiguously accessed array elements within the loop come from a different cache line. Loop interchange can help prevent this as shown in the tuned version of the code below. As shown in Figure 2-7, the data pertaining to adjacent cells in a single row will be stored in a cache line. This maximizes the possibility of re-use of data in cache memory thereby reducing the number of data calls.

It is to be noted that loop interchange might not always be achievable due to dependencies between statements. Sufficient analysis needs to be done to ensure that results do not change before this technique can be rolled into the final code.

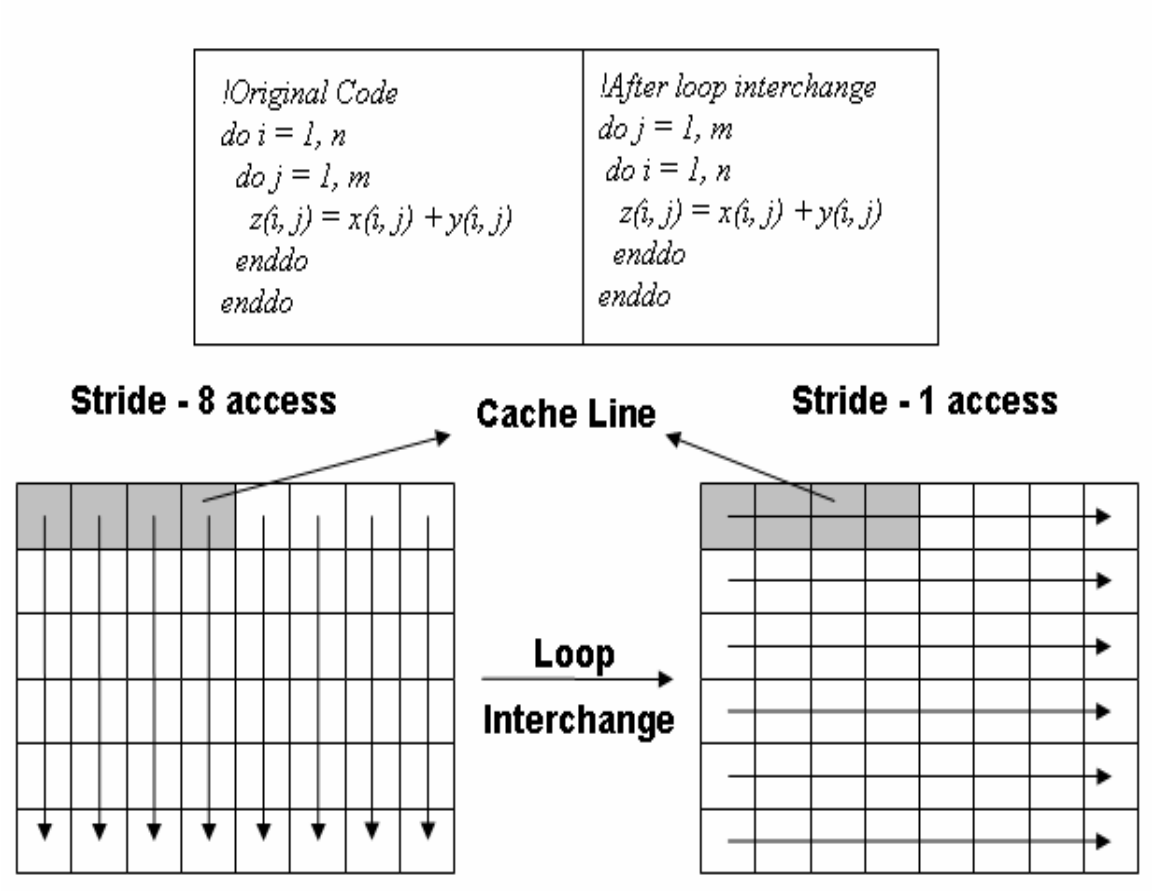


Figure 2-7 Illustration of Loop Transformation

2.6.1.3 Using Data Structures instead of arrays

This technique suggests replacing the usage of arrays by using data structures. While using arrays has its own advantages over using regular variables, sometimes using data structures in place of arrays proves to be highly efficient. Most CFD codes, like GHOST, involve lots of numerical calculations. The scenarios where addition, subtraction, multiplication or division of a variable the given point on a grid with a different variable at surrounding points on the grid are ubiquitous. This is where using data structures may prove quite useful as variables at a grid point (when declared as a data structure) are stored in contiguous memory locations thus improving the spatial locality of the program. Sufficient care has to be taken when data structures are used. For

example, the order in which variables are declared in a data structures can make a difference. This is explained with the help of two seemingly identical structures shown in Figure 2-8.

```
struct A
{
        char x;
long y;
char p;
long q;
}

struct B
{
char x1;
char y2;
long p1;
long q2;
}
```

Figure 2-8 Example to illustrate the importance of definition of data structure

The code with *Struct A* is likely to experience a decrease in performance. This is because, in memory, data is usually stored in what is called a long word, or a 32-bit word; a 4 byte unit. Longs take 4 byes; chars take 1 byte. In *Struct A*, the first char will take up the first byte of a word, and the next long, being 4 bytes, will overfill that word, so it will be allocated starting on the next long word. In memory, the way the two structures look is represented in Figure 2-11.

In the above example, while *Struct B* wastes only 2 bytes, *Struct A* wastes 6 bytes. This difference of 4 bytes, just because of order of variable declaration inside a structure is a common occurrence. The total number of bytes that would be wasted can be enormous when the size of structure is huge and especially when the code uses arrays of such structures, which is common in numerically intensive codes. Figure 2-9 presents an example in which arrays can be replaced by data structures.

```
REAL (high), DIMENSION (:,:) :: a,w,n,s,p
p(i,j) = a(i,j) + w(i,j) + n(i,j) + s(i,j) + p(i,j)
```

Figure 2-9 Arithmetic operations on arrays elements

If there are lot of occurrences in which the arrays *a, w, n, s* and *p* are being used together for calculations like the one shown above as example, it was observed, as shown

in next chapters, that data structures usage could prove to be beneficial. The example shown in Figure 2-9 can be re-written as shown in Figure 2-10.

```
TYPE struct_aewnsp
REAL (high) :: a,w,n,s,p
END TYPE struct_aewnsp
TYPE (struct_aewnsp), POINTER, DIMENSION (: ,: ) :: aewnsp
aewnsp(i,j)%p = aewnsp(i,j)%e + aewnsp(i,j)%w + aewnsp(i,j)%n +
aewnsp(i,j)%s + aewnsp(i,j)%p
```

Figure 2-10 Using data structures instead of arrays

When variables are declared inside a data structure and the code is modified to use an array of such data structures, the compiler can fetch values of the required variables at once without the possibility of data cache misses. This is because when the processor requests a set of variables, a cache line is loaded into cache and because arrays are replaced by data structures, all variables that are required in arithmetic operations are found in cache leading to fewer data misses.

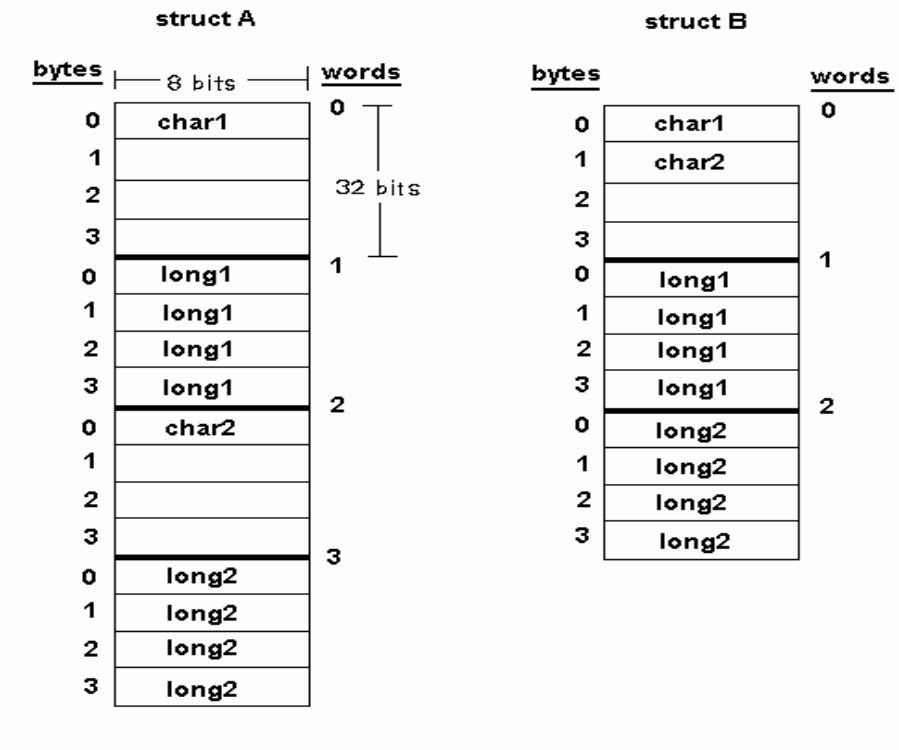


Figure 2-11 Schematic representation of memory storage for seemingly identical Structures.

2.6.1.4 Loop Blocking

This technique, also referred to as loop tiling, tends to increase the depth of a loop nest with depth *n* by adding additional loops to the loop nest [40]. This is illustrated in Figure 2-12. As mentioned earlier that the cache works based on the principle of locality of reference, the data pertaining to points surrounding a cell will be saved in the cache. Since the code traverses through the blocks, most of the data necessary will be stored in the cache in advance. This helps to improve performance by reducing cache misses.

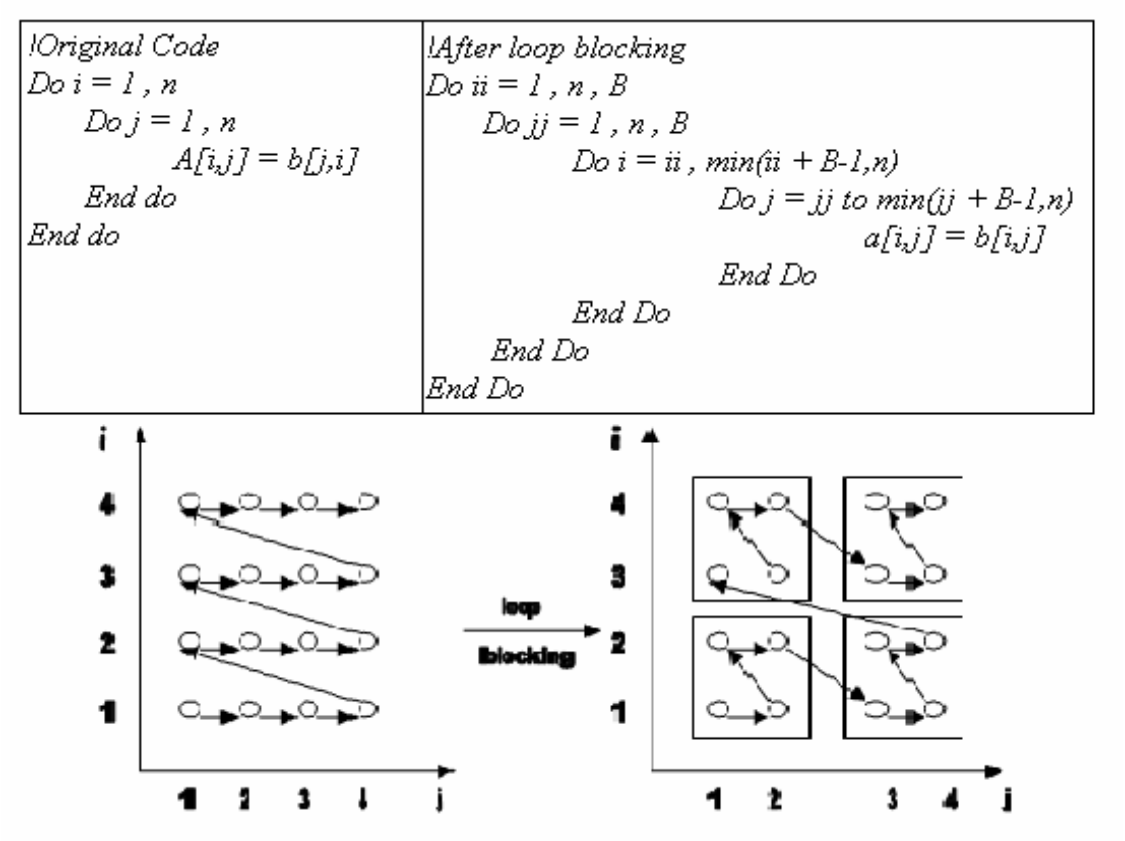


Figure 2-12 Illustration of Loop Blocking

2.6.2 OPTIMIZING FLOATING POINT OPERATIONS

In order to discern the effect of floating point optimizations, it is necessary to eliminate the limiting effects due to memory access issues. Some of the techniques presented in the earlier sections address memory access issues. In this section, a few techniques that can optimize floating point operations are discussed.

2.6.2.1 Removing Floating *If*s

'IF' statements slow down a program for several reasons. Some of them are:

- the compiler can do fewer optimizations in their presence, such as loop unrolling
- evaluation of the conditional takes time
- the continuous flow of data through the pipeline is interrupted when branching.

Often, the performance impact of '*if*' statements can be significantly reduced by restructuring the program. For example, if the result of an '*if*' statement does not change from iteration to iteration, it can be moved out of the loop. Compilers can usually do this except when loops contain calls to subroutines and when the loops are bounded by variables [40]. This is illustrated with an example in Table 2-4.

Table 2-4 Illustration of removing floating '*IF*'

!Original Code
do i = 1, lda
do j = 1, lda
if (a(i) .GT. 100) then
b(i) = a(i) - 3.7
endif
x = x + a(j) + b(i)
enddo
enddo
	!After removing floating 'if'
	do i = 1, lda
	if (a(i) .GT. 100) then
	b(i) = a(i) - 3.7
	endif
	do j = 1, lda
	x = x + a(j) + b(i)
	enddo
	enddo

2.6.2.2 Removing unwanted constants inside loops

If a constant value need not be initialized within a loop, it better to initialize it outside the loop. This is presented in table 2-5.

2.6.2.3 Avoiding Unnecessary Recalculations inside Loops

When iterating inside a loop, using pre-calculated values wherever possible instead of recalculating them every time can save a considerable amount of time. Table 2-6 illustrates this idea with an example.

Table 2-5 Illustration of removing unwanted constants inside loops

!Original Code *do i =1,n* *i1 = 0* *i2 = 0* *b(i1) = i+i1* *b(i2) = i+i2* *…….* *Enddo*	*!After removing unwanted constants* *i1=0* *i2=0* *do i = 1, n* *b(i1) = i+i1* *b(i2) = i+i2* *…….* *end do*

Table 2-6 Illustration of removing unnecessary recalculations inside loops

!Original Code *for (c = 0; p < yy; p++)* *{* *z+= x*y + p;* *}*	*!After removing unnecessary recalculations* *int n=x*y* *for (c=0; p<yy; p++)* *{ z+=n+p;* *}*

2.6.2.4 Reducing division latency by using reciprocals

In general, a floating point division operation takes a longer time (or is said to have higher latency) than multiplication or addition operations. Although division operations are infrequent than multiplication and addition operations, when their high cost is considered, even a few divisions in a code can significantly degrade its performance. For example, Flynn *et al.* [42], with experiments on their bench mark suite, describe how a relatively few number of division operations, that account for only 3% of all floating point operations, account for 40% of the latency, while multiplication operations that account for 37% of instructions contribute only to 18% of over all latency caused by arithmetic operations.

Table 2-7 Cycle times for division and multiplication operations on leading microprocessors [30]

Processor	Multiplication	Division
Intel 64-bit	12	161
AMD 64-bit	5	71

Table 2-7 gives an idea of amount of latency in machine cycles for multiplication and division operations. While multiplication requires 5 to 12 machine cycles, division latencies have a higher range and because of this great variation in latency for division operations, it is highly suggested that they are replaced by their reciprocal values especially inside a loop and nested loops thus preventing unnecessary repeated divisions. It is important to note that multiplication by a reciprocal value might sometimes alter the result. So, it is highly desirable to save the reciprocal value to the highest possible precision value before doing any subsequent operations. Many alternatives to replacing division with reciprocal have been presented over the years. For example, Oberman *et al.* [43] compares various algorithms that can be implemented in division operations in terms of their efficiency.

2.6.2.5 Subroutine Inlining

Subroutine calls incur overheads for the same reasons as loops do. To eliminate these overheads, the process of replacing the subroutine call with the function code itself is called *Subroutine Inlining*. This is particularly useful in loops with subroutine calls that have a large iteration count. Small subroutines that are called many times are good candidates for inlining. However, subroutine inlining done explicitly does not always guarantee a better performance of a code as most compilers can automatically do inlining, although this automatic compiler-inlining might not be efficient on large codes. It is advisable to avoid frequent transfer control to a subroutine, especially in loops with large iterations, as overheads of 100 cycles or more are possible for very complex subroutine calls [40]. The measured overhead on an IBM 590 for calling a simple subroutine shown in Figure 2-13 was 10 cycles [40].

```
subroutine fma(sum, x1, x2)
implicit real*8 (a-h, o-z)
sum=sum+x1*x2
return
end
```

Figure 2-13 Example to illustrate cost of calling a subroutine

However, if the subroutine to be inlined contains many lines of code, inlining can considerably increase the size of the whole code and lead to storage, readability and other performance problems. An example of subroutine inlining is illustrated in Table 2-8.

Table 2-8 Illustration of subroutine inlining

!Original Code *do 10 I=1, 1000* *call Celss (A(I),B(I))* *10 continue* *...* *subroutine celss(C,F)* *real C,F* *C = (F - 32.0) * 5.0/9.0* *return* *end*	*!After subroutine inlining* *Do 10 I=1, 1000* *A(I) = (B(I) - 32.0) * 5.0/9.0* *10 continue*

Table 2-9 Illustration of loop incorporation

!Original Code *do 10 i=1, 1000* *call celcius(x(i),y(i))* *10 continue* *subroutine celcius(c,f)* *real c,f* *c = (f - 32.0) * 5.0/9.0* *return* *end*	*!After loop incorporation* *call celciusnew(n,x,y)* *subroutine celciusnew(n,c,f)* *integer n* *real c(n), f(n)* *d0 10 i=1,n* *c(i) = (f(i) - 32.0) * 5.0/9.0* *10 continue* *return* *end*

2.6.2.7 Loop Unwinding

The technique of eliminating the inner loop by explicitly repeating the statements comprising the loop is called *Loop Unwinding*. This technique is also called *Loop Unrolling.* The idea is to reduce the number of overhead instructions that the CPU has to execute in a loop. This is illustrated in Table 2-10.

Table 2-10 Illustration of loop unwinding

!Original Code *do 10 i=1, 1000* *do 20 j=1,4* *a(i,j) = b(i,j) * 6.5* *20 continue* *10 continue*	*!After loop unwinding* *do 10 i=1, 1000* *a(i,1) = b(i,1) * 6.5* *a(i,2) = b(i,2) * 6.5* *a(i,3) = b(i,3) * 6.5* *a(i,4) = b(i,4) * 6.5*

	10 continue

2.6.2.8 Loop Defactorization

In a code with branches inside loops, in most of the cases, the body of a loop is executed in every iteration. Thus, the CPU has to do more work even where not necessary, leading to poor performance. The solution is to split the loop with a temporary array containing indices of elements to be computed on the branch (these elements usually qualified by an *IF* as shown below). This is termed as *Loop Defactorization* [40]. An example is presented in Table 2-11.

Table 2-11 Illustration of Loop Defactorization

!Original Code
do i = 1, n
if (t(i).gt.0.0) then
*a(i)=2.0*b(i-1)*
end if
end do
	!After loop defactorization
	inc = 0
	do i = 1, n
	tmp(inc) = i
	if (t(I).gt.0.0) then
	inc = inc + 1
	end if
	enddo
	do I = 1, inc
	*a(tmp(I))=2.0*b((tmp(I)-1)*
	enddo

2.6.3 OPTIMIZING FLOATING POINT OPERATIONS

On many RISC architectures, programs compiled without any optimization usually run slowly. Typically, medium optimization level (order of 2) leads to a speedup by factors of 2 to 3 [40] without a significant increase in compilation time. It is certainly worthwhile to try several optimization levels and possibly some other compiler options as well, and to assess their effect on the overall program speed.

2.7 PREVIOUS WORK

The growing dominance of parallel computational fluid dynamics has inspired greater interest in code performance and code optimization to achieve efficient use of large parallel system. This section presents some previous work done on performance tuning of various codes on wide variety of computer systems.

Douglas *et al.* [44] present optimization strategies *viz.* 2D blocking strategy, loop unrolling for both structured and unstructured grids. Speeds ups in the range of 2-5 have

been achieved on Gauss-Seidel algorithm with natural or red-black ordering on a variety of platforms. Kaushik *et al.* [45] discuss how performance tuning methodologies like field interlacing, structural blocking, edge reordering yielded improvement ratios of the order 2.26 to 6.97 on FUN3D[46] on NASA's IBM P2SC machine.

Hauser *et al* [47] discuss techniques and tools that they developed to tune their Direct Numerical Simulation (DNS) code DNSTool on a fairly inexpensive commodity cluster (cost $41205) Kentucky Linux Athlon Testbed2 (KLAT2). DNS was done on the flow over a single turbine blade and their grid was of order 16 million grid points. Their performance tuning effort involved both recoding of the application to improve cache behavior and careful engineering of cluster design. Sustained performance of 14.99 Giga Floating Point Operations per Second (GFLOPS) ($2.75 per MFLOPS) and 22.1 GLOPS ($1.86 per MFLOPS) has been achieved for double precision and single precision operations respectively.

Kadambi et al. [48] studied an algorithm to solve compressible Euler equations with regard to temporal and spatial access of data. They optimized the code by using loop interchange, reallocation of data spaces and loop fusion. They achieved a performance improvement of 45% in their best case. The L1 cache miss rate was reduced by more than a factor of four but the secondary cache miss rate did not show any significant changes.

Gropp et al. [49, 50] present optimization techniques of their CFD code FUN3D,. The first was: Interlacing, which leads to the high reuse of data brought into the cache, makes memory references closely spaced, and decreases the size of the working set of the data cache. The second was structural blocking, which lead to a significant reduction in the number of integer loads and enhanced the reuse of data in the registers. The last technique was edge and node reordering; which lead to a decrease in the TLB misses (*viz.* a kind of cache miss) by an order of two and a decrease in the L2 miss rate by a factor of 3.5. The combination of the three techniques led to an overall improvement in the execution time by a factor of 5.7.

Gupta et al. [51] and LeBeau *et al.* [52] carried out a comprehensive study of the effects of application of various cache optimizing techniques to the 3-D unstructured CFD code UNCLE. They applied space filling curve, loop blocking and optimized data

access. An overall improvement of 50% in walltime was obtained from the application of these techniques.

Palki [53] present the results of performance optimization effort of their 2-D CFD code GHOST on modern commodity clusters. Performance improvements of up to 79% were observed (when compared to original unsubblocked code) with the application of a technique called internal blocking to the unoptimized version of GHOST code. This technique involves breaking up the grid into smaller cache fitting blocks, solving the governing equations on these smaller blocks, and then putting them back together before the start of the MPI communications to get the overall solution.

CHAPTER - 3

3. COMPUTATIONAL TOOLS

This chapter presents a comprehensive description of the computational tools and platforms that have been used in this study. During the performance optimization process, it is necessary to understand the underlying algorithm in order to map it to the memory architecture of the machine on which the code runs. For this purpose, a discussion of numerics involved in the GHOST code, is presented. Linux architectures on which the optimization effort is carried on are likewise described. The latter part of the chapter consists of a discussion about *Valgrind,* a cache simulator tool. This chapter concludes with a discussion about methods used to gauge performance of the code followed by discussion about characteristics of the original version of the GHOST code.

3.1 DESCRIPTION OF GHOST

This section presents a description of original version of GHOST. It is a well established solver and has been used to carry out a number of published analyses of transitional turbomachinery flows and active flow control. Examples of recent applications of this code include the development and testing of a new laminar-turbulent transition model for turbomachinery [54, 55], simulation of an oscillatory morphing airfoil [56], the evaluation of configurations for steady jet flow control [57,58] and the simulation of plasma actuators [59].

GHOST is a two-dimensional incompressible finite-volume structured CFD code with chimera overset grids for parallel computing. The QUICK scheme is applied to discretize the advective terms in the momentum equations with second-order accuracy. A second-order central difference scheme is used for the diffusive terms. For the RANS (Reynolds Average Navier-Stokes) turbulence equations, the Total Variation Diminishing (TVD) scheme is employed for the advective terms. Interfacial fluxes are determined through interpolation of cell-centered values. Second order upwind time discretization is employed for the temporal terms, using a delta form subiterative scheme. GHOST is written in FORTRAN90 and has been ported to a wide variety of platforms. Its original version is just over 5300 lines broken into multiple subroutines. GHOST also originally designed to minimize memory usage, accomplished through extensive use of the

allocation and de-allocation of variables in FORTRAN90. GHOST uses a cell-centered partitioning approach, and the internode communication protocol is MPI. GHOST has mechanisms to do a form of automatic load balancing, but this is unnecessary for simple test geometries.

Flow and geometry data in GHOST for a given grid or subgrid are stored in individual arrays, as in $\phi_1(i,j)$, $\phi_2(i,j)$, …. $\phi_n(i,j)$. On a given grid, GHOST performs the majority of its calculations as a series of *i-j* bi-directional sweeps in nested double loops. There are more than 60 such *i-j* nested double loops in the original version of the code. For unsteady flow, second order upwind time discretization is employed for the temporal terms, which is made effectively implicit through the use of multiple subiterations within a given time step.

The momentum equations are formulated in terms of delta variables, defined as $\Delta\phi = \phi^{n+1} - \phi^n$, where ϕ represents any variable (u, v) and n is the iteration level. The resulting form of the momentum equations are solved implicitly in delta form and are shown in Eq. (3-1) for the time discretization in one dimension:

$$\frac{3(\Delta\phi)^m}{2\Delta t} + \frac{\partial f(\Delta\phi)^m}{\partial x} = \frac{(\phi^n - \phi^{n-1})}{2\Delta t} - \frac{3((\phi^{n+1})^m - \phi^n)}{2\Delta t} - \frac{\partial f((\phi^{n+1})^m)}{\partial x}, \qquad (3\text{-}1)$$

where m is the subiteration level. The right-hand side of Eq. (3-1) is explicit and can be implemented in a straightforward manner to discretize the spatial derivative term. The left-hand side terms are evaluated based on the first order upwind differencing scheme. The deferred iterative algorithm is strongly stable, and the solution ϕ^{n+1} is obtained by using inner iterations to reach the convergent solution of the right-hand side of Eq. (3-1), corresponding to $\Delta\phi$ approaching zero. At least one subiteration is performed at every time step so that this method is fully implicit. For steady flows, Δt is set to infinity and convergence is achieved through the subiteration cycle.

The resulting matrices generated at each subiteration based on the QUICK and TVD schemes as well as evaluation of source/sink terms are solved with ADI-type decomposition into a pair of sweeps alternately in the *i*- and *j*-directions which are solved sequentially in tri-diagonal matrices. This sequence may be repeated for improved accuracy. The techniques of Rhie and Chow [60] are then used to extract the pressure field from the continuity equation. Other equations, such as energy conservation and

turbulence models, are computed in turn as necessary. Then, the subiteration sequence is repeated until satisfactory convergence is achieved.

For clarity most of the performance testing was conducted on the simplest form of GHOST that has only the steady-state version of the code, laminar model only (no turbulent flow), and only a single pair (one in *i*, one in *j*) of ADI computations is completed per iteration. However, the iterative core of the code is retained even in this simplified version.

There are many different schemes that can be employed besides those implemented in GHOST, but many CFD codes follow similar procedures that require discretization of the conservation equations and then solving the system of equations for the grid. Therefore, the GHOST code that is analyzed here can be considered a representative sample of this common type of CFD algorithm.

3.1.1 GHOST FLOW CHART

The underlying algorithm in GHOST code was described in fair amount of detail in previous section. This section describes the flow chart of GHOST code that is shown in Figure 3-1.

The first task that is performed by the code is to find out the number of grid files to be read and the number of processors to be used in solving the problem. This is done in subroutine *read_map*. The code reads this information from the file called *mpi.in*. Contents of the file *mpi.in* are shown in Figure 3-2.

The second task performed by the code is to read the grid files. This is done in subroutine *read_data*. Flow field variables (*viz. u, v*) and boundary conditions are initialized next. In case the grids are located on different processors, the boundary conditions are communicated between the grids by using MPI broadcasts. Before the code starts the calculations, it checks to see if there are any restart files present. This is done by subroutine *read_restart*. Restart files mainly contain the *u, v* and *p* values at each of the grid points from the previous run.

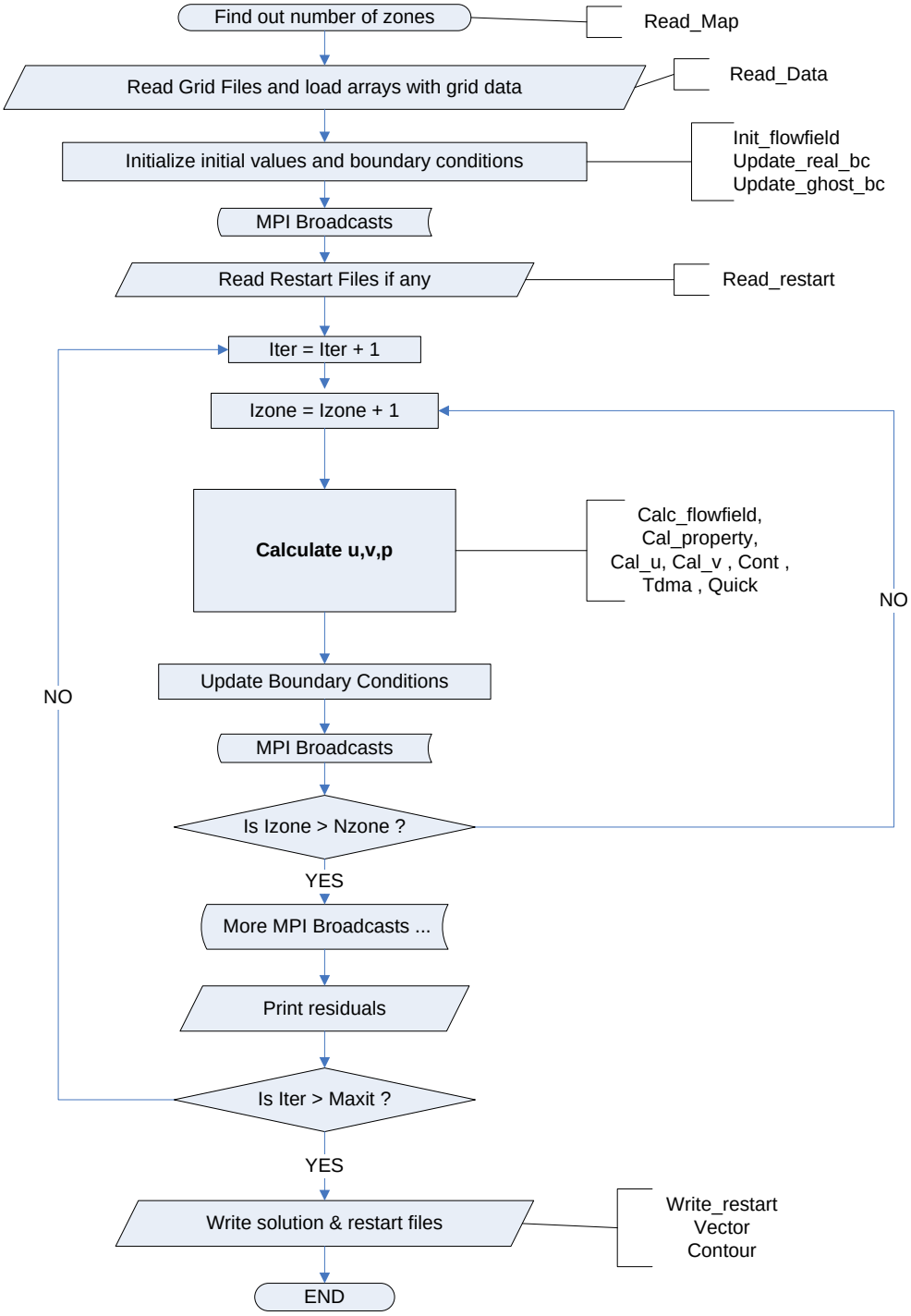


Figure 3-1 Flowchart depicting the working of GHOST [53]

```
1 2 3 4 5 6 7 8 !Zones
0 1 1 2 1 2 3 3 !Nodes
```

Figure 3-2 Contents of the file mpi.in

Flow calculations that were mentioned in the previous paragraph are carried out in subroutine *calc_flowfield*. This subroutine initially calculates the variables that are necessary to solve the momentum and continuity equations. These equations are actually solved in subroutine c*al_property.*

The next task performed by the code is to update the *u* velocity by solving the *x*-momentum equation. This is done in subroutine *cal_u*. Subroutine *quick* initially applies the QUICK scheme and calculates the required parameters. Subroutine *tdma* then solves the tri-diagonal matrix that is formed. This is done using the standard TDMA method.

In a similar way, the *y*-momentum equation is solved to get the *v* velocity field (subroutines *cal_v, quick, tdma*). Once the *x* and *y* momentum equations have been solved and the velocity field has been obtained, the pressure field is extracted from the continuity equation using the Rhie and Chow technique [60] (subroutines *cont, quick, tdma*).

If the energy equation is switched on then an additional subroutine is used to calculate the temperature field (subroutine *cal_t*). If the turbulence model is switched on, two additional subroutines are used to calculate the eddy viscosity (subroutines *cal_tk, cal_ed*). Once all the required flow field variables have been calculated, the code broadcasts these values to all the processors since they are required to calculate the values at the boundary points and to update them.

After the values at boundaries points are calculated, the code once again broadcasts these newly calculated values. It then calculates the residuals and prints them. If the solution is converged, the solution is written to a file, if not it starts off with next iteration until either the solution converges or the iteration number is greater than the *maxit* value in the code. A summary of what each subroutine does is shown in Table 3-1.

Table 3-1 Summary of subroutines in original version of GHOST

Subroutine	Function
Cal_u	Solves the *x*-momentum equation to update the *u* - velocity field.
Cal_v	Solves the *y*-momentum equation to update the *v* - velocity field.
Cont	Extracts pressure field from continuity equation.
tdma	Tri diagonal matrix solver
cal_property	Computes the values of variables necessary to solve governing

	equations.
quick	Implements the QUICK scheme to determine the advective fluxes.
cal_t	Solves the energy equation to calculate the temperature field.
cal_tk	Computes the turbulent kinetic energy (k)
cal_ed	Computes the energy dissipation rate (ε)

3.1.2 GOVERNING EQUATIONS

The governing equations for unsteady incompressible viscous flow under the assumption of no body force and heat transfer that are used to calculate the various flow field parameters in GHOST are as below:

Conservation of Mass

$$\frac{\partial}{\partial t}\int_V \rho\, dV = -\oint_S \rho u_i n_i dS \ ; \tag{3-2}$$

Conservation of Momentum

$$\frac{\partial}{\partial t}\int_V \rho u_j dV = -\oint_S \rho u_i n_i \ u_j dS - \oint_S p n_j dS + \oint_S \tau_{ij} \ n_i dS \ ; \tag{3-3}$$

Conservation of Energy

$$\frac{\partial}{\partial t}\int_V \rho E dV = -\oint_S \rho u_i n_i \ E dS - \oint_S p u_j n_j dS + \oint_S u_j \ \tau_{ij} n_i dS \ ; \tag{3-4}$$

where ρ is density, p is pressure, u_i are the components of the velocity vector, n_i is unit normal vector of the interface, τ_{ij} is tensor of shear force, and specific energy is $E = e + \frac{1}{2}(u^2 + v^2 + w^2)$.

3.1.3 CALCULATION AT ARTIFICIAL BOUNDARIES

GHOST uses the chimera overset grid [63] method to carry out parallel computations. This technique is used to carry out calculation at artificial boundaries that are formed due to splitting the grid into smaller blocks. This is done to carry out parallel computations. This section briefly explains this technique.

Figure 3-3 shows a grid that has been split into two halves for the sake of performing parallel computations. A magnified view of the region of overlap is shown below the actual grid. As can be noticed in the Figure 3-3, four grid points from each zone are overlapped. The last two overlapped grid points for each zone are referred to as

"Ghost Points". No calculations are carried out at the Ghost Points. The number of Ghost Points required depends upon the order of accuracy of the code. GHOST is second order accurate; so it uses information from two grid points in each direction surrounding the grid point while calculating the diffusive and convective fluxes. Hence it requires two Ghost Points at the artificial boundaries.

Assume that each of these zones is located on two separate nodes of a parallel computer. For the first iteration, the code performs the calculation on each of the nodes simultaneously using the initial values and real boundary conditions. There is no transfer of data between the nodes during this stage (This capability has been used to implement internal blocking [53]). At the end of the first iteration, using MPI communication, the boundary information is transferred between the nodes. For a laminar case with no heat generation, the values of velocities (u, v) and the pressure (p) are swapped between the nodes. The values at the boundary points of zone 1, *i.e.* W_{n-1} and W_n , are passed on to the ghost points of zone 2, *i.e.* E_0 and E_1 and the same is done between the ghost points of zone 1 and boundary points of zone – 2. During the second iteration, the ghost point values are used to calculate the values at the boundary points. The same process is followed at the end of each iteration.

3.2 COMPUTATIONAL GRID

3.2.1 FINITE VOLUME METHOD

GHOST uses the finite volume method to solve the governing flow equations. This section presents a brief introduction to this technique. "Finite volume" refers to the small control volume surrounding each node point on a mesh. The governing integral equations are enforced on this control volume. A typical control volume (CV), along with the notations is shown in the Figure 3-4.

The control volume consists of four faces, denoted by lower-case letters (*n, e, w, s*) corresponding to their location with respect to the central node C. Adjacent nodes are denoted by upper case letters (*i.e. N, E, W, S*). The values of the flow variables are calculated and stored at the cell centers *i.e.* nodes. The vertices around the central node C are denoted by lower-case letters (*ne, nw, sw, se*). The values of the flow variables at the vertices are calculated by taking the weighted average of the values at the nodes (*N, E, W and S*) surrounding the vertex.

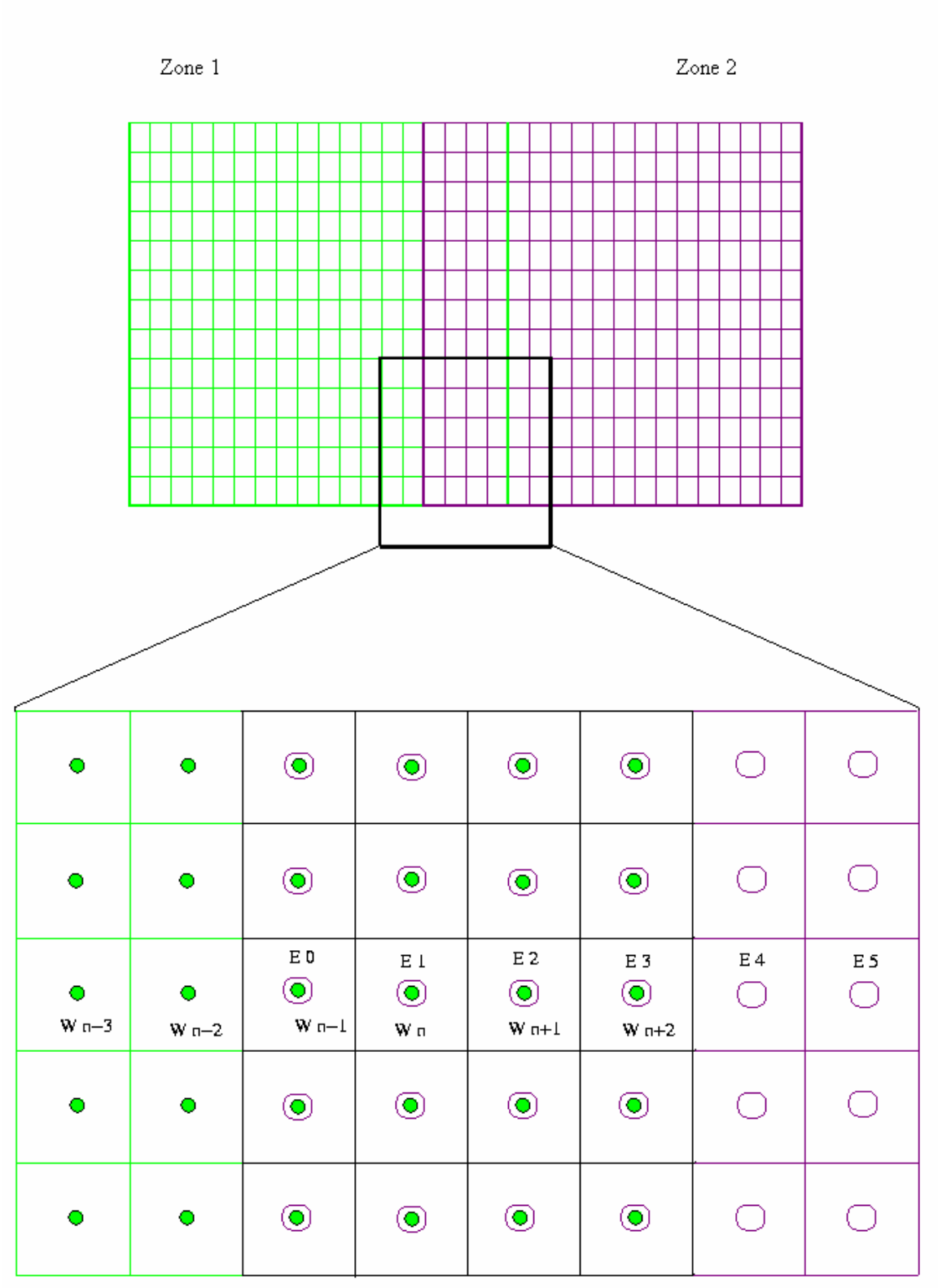


Figure 3-3 Illustration of artificial boundaries

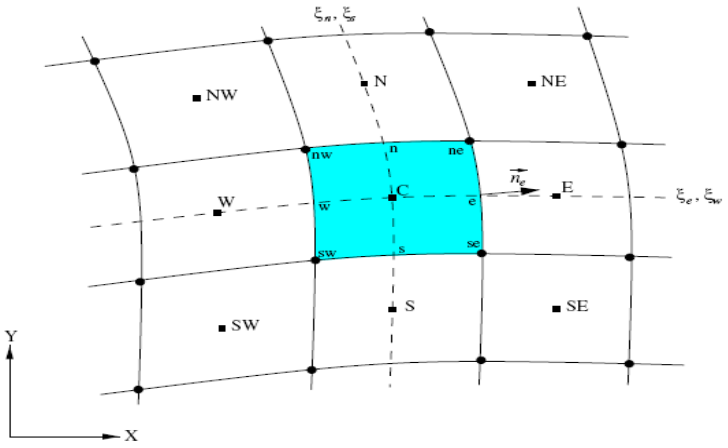


Figure 3-4 A grid in generalized coordinate system [64]

3.2.2 GRID FILES

Since GHOST works off a generalized coordinate system, it requires a lot of grid data apart from just the *x, y* co-ordinates of the grid points. This data is generated by the code g.f90. In this section we briefly describe the grid data that is generated by this code or the contents of the grid file for a non moving grid are as follows:

- Number of ghost points,
- Grid point weight in the *x* and *y* direction,
- *x* and *y* co-ordinates of the grid points,
- Volume of the cell surrounding each grid point,
- Distance between the wall and the grid point,
- Values for the various transformation functions such as η_x, η_y, ξ_x and ξ_y,
- A variable called "*inx*" which specifies if a particular grid point is a ghost point or not. If the value of *inx* for a grid point is 1, then that particular grid point is treated as a ghost point, whereas if its zero, it is treated as a normal point.
- Boundary conditions.

GHOST reads all these flow and geometry data of the computational domain from the grid files and stores them in a single structure consisting of various arrays for each of the above mentioned data.

3.2.3 DESCRIPTION OF INPUT FILE

As mentioned in the previous section, g.f90 is used to generate the grid data required by GHOST. In order for g.f90 to generate a grid it requires certain data regarding the size of the computational grid, boundary conditions and number of grid points. This data is provided using the file called "input". An *input* file to generate a 200x200 grid (split into 4 blocks) is shown in Figure 3-5, with explanations inline. Under the column "Patch Zone Number" the value for the row labeled right (in zone 1) is 2, which means that the zone 2 is to the right of zone1. Similarly in the row labeled left in zone 2, we have specified the value as 1, which means that the zone 1 is to the left of zone 2. Input file to generate a single zone 100 x 100 grid is shown in Figure 3.6 and is simpler than the one for multiple zones.

```
 /*number_of_zone
  4
 /* zone_number 1
 quadratic 100 100  4 2
(!Type of grid , No. of point in x direction , y direction , No. of Boundary Conditions,
No. of Ghost Points)
     0.00     0.00                              (!Center co-ordinates)
     0.00     0.50     0.50     0.00           (!X -co-ordinates of corners of the grid)
     0.00     0.00     0.50     0.50           (!Y -co-ordinates of corners of the grid)
     0.00     0.00     0.00     0.00           (!Wall co-ordinates)
     0.99      1.0                             (!Ratio to specify the grid density.)
[ If < 1 (Eg. 0.99) then the grid density INCREASES from left to right, ]
[If > 1 (Eg. 1.04) then the grid density DECREASES from left to right.]
 (! Boundary Conditions Type of Boundary,  Relative Position,  Patch zone number)
 * wall            left    1      1      1     99999  0
 * wall           top    -99999   100000  99999  99999  0
 * patch           right   99999   99999   1     99999  2
 * wall           bottom  -99999   100000  1     1      0
 /* zone_number 2
 quadratic 100 100  4 2
     0.00     0.00
     0.50     1.00     1.00     0.50
     0.00     0.00     0.50     0.50
     0.00     0.00     0.00     0.00
     1.0      1.0
 * patch           left    1      1      1     99999  1
 * wall           top    -99999   100000  99999  99999  0
 * wall           right   99999   99999   1     99999  0
 * wall           bottom  -99999   100000  1     1      0
 /* zone_number 3
 quadratic 100 100  4 2
```

```
      0.00     0.00
      0.50     1.00     1.00     0.50
      0.50     0.50     1.00     1.00
      0.00     0.00     0.00     0.00
      1.0      1.0
 * patch            left    1      1      1     99999  4
 m inlet            top     -99999   100000  99999  99999  0
 * wall            right   99999   99999   1     99999  0
 * patch            bottom  -99999   100000  1     1     2
  /* zone_number 4
  quadratic 100 100  4 2
      0.00     0.00
      0.00     0.50     0.50     0.00
      0.50     0.50     1.00     1.00
      0.00     0.00     0.00     0.00
      1.0      1.0
 * wall            left    1      1      1     99999  0
 m inlet            top     -99999   100000  99999  99999  0
 * patch            right   99999   99999   1     99999  3
 * patch            bottom  -99999   100000  1     1     1
 /end
```

Figure 3-5 Description of input file for 4 a zone grid

```
  /*number_of_zone
   1
  /* zone_number 1
  quadratic 100 100  4 2
      0.00     0.00
      0.00     1.00     1.00     0.00
      0.00     0.00     0.50     0.50
      0.00     0.00     0.00     0.00
      0.99      1.0
* wall            left    1      1      1     99999  0
m inlet            top     -99999   100000  99999  99999  0
 * wall            right   99999   99999   1     99999  0
 * wall            bottom  -99999   100000  1     1     0
/end
```

Figure 3-6 Description of input file for a 1 zone grid

3.3 COMPILERS and MPI ENVIRONMENT

The Intel FORTRAN Compiler and g95 FORTRAN compiler were used to compile the code for this work. Since GHOST is an MPI based code, an MPI environment has to be installed on the machines for it to be compiled and run. LAM/MPI has been used for this purpose. Compilers that have been used in the present work are Intel FORTRAN Compiler Ver. 7.1 (Ifc), and the FORTRAN Compiler Ver. 9 (Ifort)

LAM/MPI [65] was originally developed at the Ohio Supercomputing Center. It is a high quality implementation of the Message Passing Interface (MPI) Standard. LAM/MPI provides high performance on a variety of platforms, from small off-the-shelf single CPU clusters to large SMP machines with high speed networks. In addition to high performance LAM provides a number of usability features key to developing large scale MPI applications. MPICH a freely available, portable implementation of MPI and is used on KFC6A (described later).

3.4 PROFILING TOOLS

In recent years, with the advent of memory debuggers and profilers, it has become relatively easier to identify bottlenecks in a given code. These kinds of tools are particularly useful while working on performance optimization of CFD codes. *Valgrind* [61] is one such memory debugger and profiler that has been used in this work. With Valgrind's tool suite, a programmer can automatically detect many memory management and threading bugs, making programs more stable. Detailed profiling can also be done to help speed up the programs. Valgrind is an open source tool and it does not require the user to recompile, relink, or modify the source code. On the other hand it has the disadvantage of slower runtime. This is usually justified keeping in view the time that is saved once the code is optimized.

When tuning a code, it is advised to optimize the largest bottleneck first. With the help of *Valgrind,* we will be able to identify how much time is being spent on each of the subroutines as shown as an example in Table 3-2 below:

Table 3-2 Illustration of Valgrind output

PROCEDURE	TIME
main()	13%
subroutine1	17%
subroutine2	20%
subroutine3	50%

Profiling output from Valgrind, like the one shown in Table 3-2 empowers the researcher with a good starting point. In the above example, when a tuning effort is begun with subroutine3, for instance, a 20% decrease in its time will yield an overall 10% increase in performance while on the other hand, a 20% decrease in main() will yield only an overall 2.6% increase in performance. Thus, as discussed above, *Valgrind* aids programmers and scientists with a good starting point and an overall map for the tuning effort. Some of the benefits associated with Valgrind are that:

- Uses dynamic binary translation so that modification, recompilation or relinking of the source code is not necessary;
- Debugs and profiles large and complex codes;
- Can be used on any kind of code written in any language;
- Works with the entire code, including the libraries;
- Can be used with other tools, such as GDB;
- Serves as a platform for writing and testing new debugging tools.

The *Valgrind* suite comprises of five major tools Memcheck, Addrcheck, Cachegrind, Massif, and Helgrind which are tightly integrated into the Valgrind core.

Memcheck checks for the use of uninitialized memory and all memory reads and writes. All the calls to malloc, free and delete are instrumented when memcheck is run. It immediately reports the error as it happens, with the line number in the source code if possible. The function stack tracing tells us how the error line was reached. The tracks are addressed at byte level and initialization of values is addressed at bit level. This helps Valgrind detect the non-initialization of even a single unused bit and note report spurious errors on bitfield operations. The drawback of memcheck is that it makes the program run 10 to 30 times slower than normal.

Addrcheck is a toned down version of Memcheck. Unlike Memcheck it does not check for uninitialized data, which leads to Addrcheck detecting fewer errors than Memcheck. On the brighter side it runs approximately twice as fast (5 to 20 times than normal) and uses less memory. This allows the programs to run for longer time and cover more test scenarios. In summation, Addrcheck should be run lo locate major memory bugs while Memcheck should be used to do a thorough analysis.

Massif is a heap profiler. The detailed heap profiling is done by taking snapshots of the program's heap. It produces a graph showing heap usage over time. It also provides information about the parts of the code that are responsible for the most memory allocations. The graph is complemented by a text or HTML file that includes information about determining where the most memory is being allocated. Massif makes the program run approximately 20 times slower than the normal.

Helgrind is a thread debugger. It finds data races in multithreaded codes. It searches for the memory locations which are accessed by more than one thread but for which no consistently used lock can be found. These locations indicate of loss of synchronization between threads and could potentially cause timing-dependent problems.

3.4.1 CACHEGRIND

Cachegrind is a cache profiler. It performs detailed simulation of the L1, D1, and L2 caches in a CPU. It helps in accurately pinpointing the sources of cache misses in the source code. It provides the number of cache misses, memory references, and instructions executed for each line of source code. It also provides per-function, per-module, and whole-program summaries. The programs run approximately 20 to 100 times slower than normal run times. With the help of the KCacheGrind [62] visualization tool, these profiling results can be seen in a graphical form which is easier to comprehend. Cachegrind has been extensively used in this study.

Once the code is compiled and the executable file is generated, cachegrind can be run on the executable file to analyze the cache behavior of the code. If, for example, *tempsstnt* is the executable code for GHOST code, cache analysis using cachegrind is done at the command prompt as shown below:

$ mpirun –np n valgrind –skin=cachegrind tempsstnt

where n = number of processors. A sample output of the above command is shown in Figure 3-7

In addition to the above shown output, cachegrind also provides the number of cache misses (both instruction and data), instruction references, and data references at subroutine level. This information is critical in understanding how a particular change in the code translated to change in cache behavior at subroutine level. This is explained in detail in chapter 4.

```
I   refs:      4,284,365,937
I1  misses:          540,146
L2i misses:          287,164
I1  miss rate:           0.1%
L2i miss rate:           0.0%
D   refs:      2,622,017,888  (2,074,442,953 rd + 547,574,935 wr)
D1  misses:      271,115,024  (  233,005,237 rd +  38,109,787 wr)
L2d misses:       32,543,188  (   26,585,763 rd +   5,957,425 wr)
D1  miss rate:          10.3% (         11.2%   +         6.9%  )
L2d miss rate:           1.2% (          1.2%   +         1.0%  )
L2 refs:         271,655,170  (  233,545,383 rd +  38,109,787 wr)
L2 misses:        32,830,352  (   26,872,927 rd +   5,957,425 wr)
L2 miss rate:            0.4% (          0.4%   +         1.0%  )
```

Figure 3-7 Sample output from cachegrind

In the Figure 3-7,

I refs = Instructions executed

I1 misses = instruction read misses in L1 cache memory

L2i misses = instruction read misses in L2 cache memory

I1 miss rate = instruction miss rate on L1 cache memory

L2i miss rate = instruction miss rate on L2 cache memory

D refs = Sum of data cache reads (i.e., memory reads) and data cache writes (i.e., memory writes)

D1 misses = data misses on L1 cache memory (D1 misses = sum of L1 data read and L1 data write misses)

L2d misses = data misses on L2 cache memory (L2d misses = sum of L2 data read and L2 data write misses)

L2 refs = number of references to L2 cache (L2 refs = sum of L2 data read and L2 data write references)

Valgrind's cache profiling has a number of shortcomings [61]

- It does not account for kernel activity -- the effect of system calls on the cache contents is ignored.

- It does not account for other process activity (although this is probably desirable when considering a single program).
- It does not account for cache misses that are not visible at the instruction level, eg. those arising from TLB misses, or speculative execution.
- Valgrind's custom threads implementation will schedule threads differently to the standard one. This could warp the results for threaded programs. This should only happen rarely.
- FPU instructions with data sizes of 28 and 108 bytes (e.g. `fsave`) are treated as though they only access 16 bytes. These instructions seem to be rare so hopefully this will not affect accuracy much.

3.5 KENTUCKY FLUID CLUSTERS

This section has brief technical configuration information about different clusters on which the performance optimization effort has been carried out. These clusters are housed at Department of Mechanical Engineering, University of Kentucky. The current work is focused on optimizing the CFD code GHOST on Kentucky Fluid Clusters 3, 4, 5 and 6A and 6I. Kentucky Fluid Clusters 3, 4 and 5 (KFC3, KFC4 and KFC5) are shown in Figure 3-8.

The Intel FORTRAN90 compiler (ifort) with -O3 optimization, the G95 compiler also with -O3 optimization, LAM MPI, and MPICH were used for the purpose of compiling GHOST for this study. Since these clusters are controlled in-house, nodes can be readily restricted to a single job at a time; as such, the difference between the CPU time and the walltime has proven negligible, so walltime is used as the basis of the testing. Time values also exclude I/O.

Kentucky Fluid Cluster 3 consists of fifteen 2.4 GHz Pentium 4 nodes and one 3.0 GHz Pentium 4 server/node linked by a 16-port commodity Gigabit switch. Each node has 512 MB of RAM (2 GB on the server) and each processor has a L2 cache of 512 KB. KFC3 nodes are off-the-shelf Dell PCs, resulting in a minimum of hardware construction in exchange for a higher per node cost.

Figure 3-8 Kentucky Fluid Clusters (KFC) 3, 4 and 5

Kentucky Fluid Cluster 4 is constructed with AMD Athlon 2500+ 1.826 Ghz 32 bit Barton processors. The current configuration is a 47 node system linked by two networks: a single Fast Ethernet (100 Mb/s) switch and a single Gigabit (1Gb/s) switch. Each node has 512 MB of RAM and each processor has a L2 cache of 512 KB. The server is separate from the nodes and plays no direct role in the iterative computation. KFC4 is housed at the University of Kentucky.

Kentucky Fluid Cluster 5 is a 64-bit architecture, constructed of 47 AMD64 2.08 GHz processors linked by a single Gigabit (1Gb/s) switch. Each node has 512 MB of RAM and each processor has a L2 cache of 512 KB. The server is separate from the nodes and plays no direct role in the iterative computation. Like KFC4, KFC5 is housed at the University of Kentucky.

Kentucky Fluid Clusters 6A and 6I are two similar clusters based on dual-core processors. Each node has 1 GB of memory and the nodes are linked by a Gigabit switch.

KFC6A has 23 4600+ AMD Athlon 64 X2 dual core processors at 2.4 GHz and 512 KB x2 L2 cache. KFC6I has 24 Intel Core 2 Duo E6400 processors running at 2.13 GHz and with 2 MB L2 cache. In addition, as part of the cluster design processes a single workstation with a 4200+ AMD Athlon 64 X2 dual core processor has been constructed and used for testing. The critical difference between the 4200 and 4600 is a lower processor speed (2.2 GHz). Details of these processors that are relevant to the current work are given in the Table 3-3.

Table 3-3 Comparison of the KFC6 processors based on certain parameters

Processor	Clock Speed	L1 Cache Size	L2 Cache Size	FSB
Intel E-6400	2.13 GHz	2 X 32 Kb	1 X 2 Mb	1066 MHz
AMD 4200+	2.2 GHz	2 X 128 Kb	2 X 512 Kb	2000 MHz
AMD 4600+	2.4 Ghz	2 X 128 Kb	2 X 512 Kb	2000 MHz

3.6 METHODS USED TO MEASURE PERFORMANCE

In the computing world two ways of measuring the time taken by a code are referred to as “wall clock time” and “CPU time” [66]. Wall clock time is the time that passes if you are looking at a clock on the wall for the code to finish a problem. CPU time is the amount of time spent by the CPU in carrying out the calculations. The CPU time excludes time for events such as passing the data across the network, I/O time, CPU interrupt time and processing TCP packets. All these usually affect the total time taken to complete the job. Hence CPU time can miss critical time costs for someone doing CFD runs on parallel systems. CPU time can be calculated by calling cpu_time function.

In the present work, wall clock time is used to measure the code performance improvements in spite of running all our tests on a single node. All our tests have been carried out on clusters that are controlled in-house. Hence it was seen to it that only a single job is running on the node while carrying out the timing tests. Based on the initial tests that were conducted, it was noticed that the difference between the walltime and CPU time was minimal under these conditions.

Walltime is the sum of the time taken by the code to read the grid files and to write the solution (I/O time) and time taken to complete the calculations (Solver time). Since this work is concerned with the optimization of the solver portion of the job only, the solver time alone was considered while calculating the performance improvements gained, which meant I/O time was not considered as the walltime. So, a reference to walltime from now on actually means walltime excluding I/O. Walltime is calculated by using mpi_wtime() function. This function returns a floating-point number of seconds, representing elapsed wall clock time since some time in the past. This function is called at appropriate places in the code and walltime is calculated by difference of two consecutive calls.

For a given problem and a code, walltime is a function of the grid size and the number of iterations. If the number of iterations is kept constant and the grid size increased, the walltime will increase too. In order to compare the performance improvements obtained with varying grid sizes, the walltime has been normalized by the grid size and number of iterations. Hence the walltime that has been used to measure the performance improvement is approximately the wall clock time of the code to perform a single iteration on a single grid point. This is further explained in detail in chapter 4.

3.7 EXTERNAL BLOCKING

This technique has been widely used to carry out multi-node performance tests that are described in chapter 4. External blocking involves the breaking up of the computational grid into smaller sized cache friendly blocks (Figure 3-9) so that these blocks can be solved on more than one node instead of solving the entire grid on one node. This step is carried out during the grid generation process. The only difference in the process of generating the grid with a single block vs. multiple blocks is that the content of *input* file has to reflect the details in either case. This has been described in section 3.2.3.

3.8 CHARACTERISTICS OF ORIGINAL CODE

Before we delve into the details of tuning process (presented in chapter 4), it is important to understand the characteristics of the original version of the GHOST code (V0). In this section, performance behavior of V0 is presented with reference to the

cavity flow problem as test case. The behavior might vary based on hardware and the type of computational problem, but relative results are consistent if a sufficiently large grid and large number of iterations are used.

As described in earlier sections of this chapter, the original version of the GHOST was designed to minimize memory usage and this is accomplished through extensive use of the allocation and de-allocation of variables in FORTRAN90. As with many numerically intensive codes, GHOST is no exception to the 80-20 rule. More than 98 percent of the computing time is spent in six subroutines for a reasonable grid size and in laminar flow conditions. Table 3-4

Approximate computing time spent in each subroutine is shown in Table 3-4. These subroutines represent the sub iteration cycle for a two-dimensional, laminar flow.

Table 3-4 Approximate percentage of time spent in each subroutine in V0 for a 2-D cavity laminar flow

module::subroutine	**% of total computing time**
calc::cont	40%
calc::tdma	20%
calc::cal_property	19%
Global::quick	7%
calc::cal_v	6%
calc::cal_u	5%

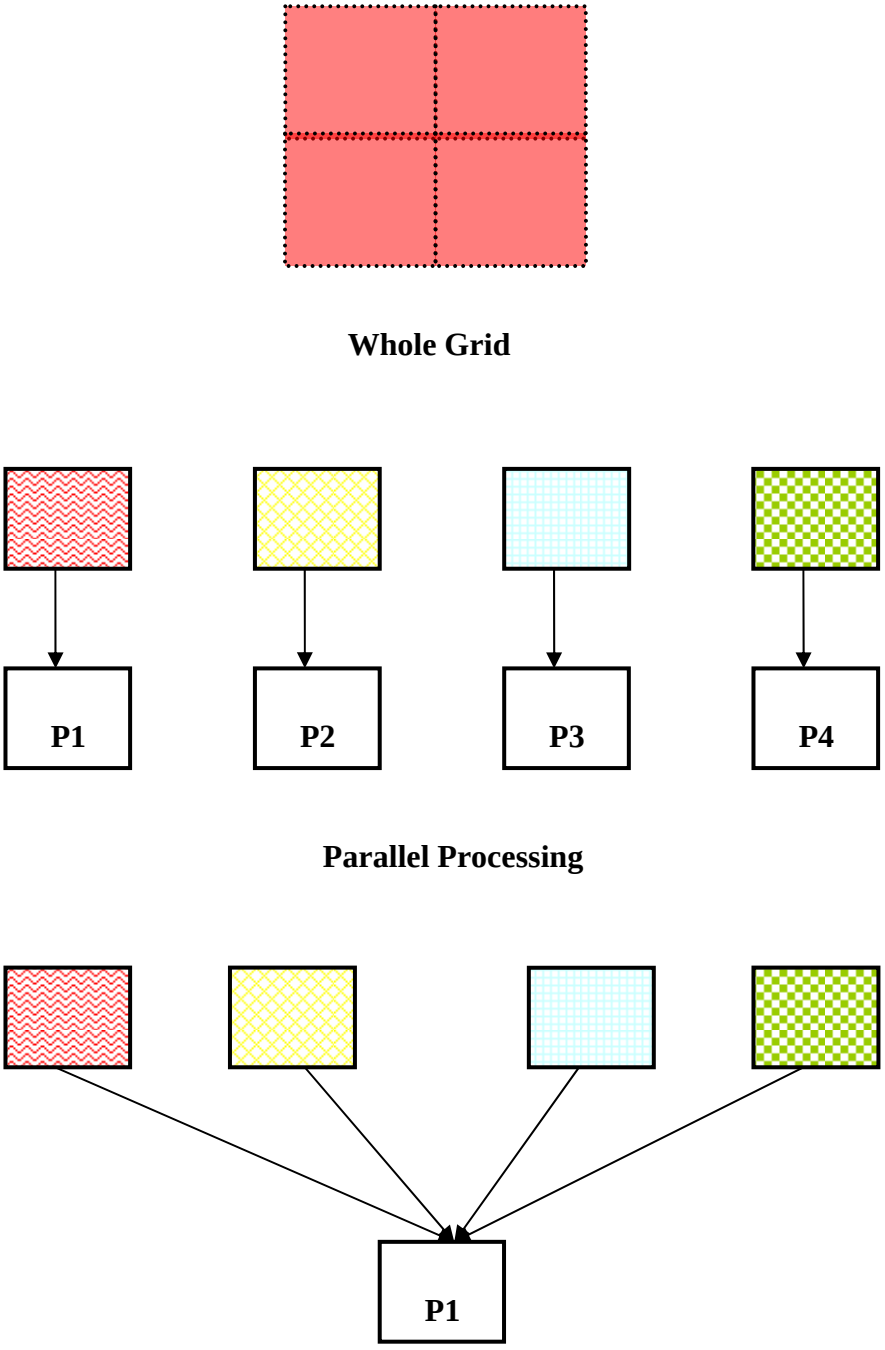


Figure 3-9 External blocking

3.8.1 DETAILS OF CRITICAL SUBROUTINES

This section presents important details of the subroutines in Table 3-4 in V0. Although there is no direct correlation between the size of a code and the time taken to run, details of the size of each subroutine are presented in the discussion for clarity.

- Subroutine *Cont*: Out of 5300 lines of V0, this subroutine is not more than 240 lines long but accounts to 40% of total computation time as presented in Table 3-4 i.e., approximately 4.5% of the code taking up to 40% of the computation time. The number of calls to this subroutine is equal to the number of iterations. This subroutine has 8 nested loops that span across the dimensions of the grid. The following details have been observed in these nested loops:
    - 5 out of the 8 nested loops have their sweeps in *i, j* order. However, this is the not the physical order in memory according to FORTRAN convention as discussed in chapter-1. For example, in the nested loop presented in the Figure 3-10, the elements *au(i,j) and au(i+1,j)* are actually *nj* addresses apart, and thus a cache miss will occur not only on the first *nj* loads of this array *au* but also the order of the mismatch between the order of data accesses and data storage in FORTRAN leads to repeated cache misses.

```
DO i = 0, ni
   DO j = 1, nj
      auu = 1. / au (i, j) + rp * (1./au(i+1, j)-1./au(i, j))
      ……..
   END DO
END DO
```

Figure 3-10 Example of mismatch between data access and data storage

    - Reciprocal values of variables *au* and *av* are being repeatedly calculated inside *i-j* sweeps. For example, the reciprocal of variable *au* is being calculated 26 times in one sweep along *y*-direction. As discussed in chapter 1, division operations considerably take more cycles than any other arithmetic operation. Added to this, the fact that these reciprocals are being calculated inside nested *do* loop whose sweep does not coincide with physical order in memory compounds this problem and leads to unnecessary cycles and heavy cache misses.
- Subroutine *tdma:* Out of 5300 lines of V0, this subroutine is not more than 110 lines long but accounts to 20% of total computation time as presented in the Table 3-4 i.e., approximately 2.0% of the code taking up to 20% of the computation time. This subroutine solves the tri-diagonal system of equations using the TDMA method [67].

The high cost of this subroutine is attributed to the fact that calculations like the one shown in Figure 3-11 compel the processor to make repetitive reference to elements of arrays *viz ae, aw, an, as, ap*.

```
res (i,j) = rhs (i,j) - ap (i,j) * du (i,j)
              + aw (i,j) * du (i-1,j)
              + ae (i,j) * du (i+1,j)
              + as (i,j) * du (i,j-1)
              + an (i,j) * du (i,j+1)
```

Figure 3-11 Example of repetitive reference to array elements in GHOST

Due to their size, these arrays cannot be fit into cache and so will lead to repetitive cache misses. An attempt has been made to replace arrays with arrays of data structures with an aim of addressing this problem. This is discussed in chapter 4. This subroutine also has a nested loop in which the data access does not match with the data storage in FORTRAN. The orientation of *i, j* loop cannot be reversed as such a change will change the algorithm of this subroutine. Also, the problem is compounded by the fact that, in every iteration, this subroutine is called by three subroutines *viz. cal_u, cal_v and cont*. This means, the total number of calls to this subroutine is three times the number of iterations.

- Subroutine *cal_property*: Out of 5300 lines of V0, the part of this subroutine that deals with laminar flow is approximately 150 lines long. This accounts to approximately 19% of total computation time as presented in Table 3-4 i.e., approximately 2.8% of the code taking up 19% of the computation time. The high cost of this subroutine can be attributed to the fact that out of three nested *do* loops that span across the dimensions of the grid, two of them have their sweeps in *i, j* order. However, this is the not the physical order in memory according to FORTRAN convention as discussed in chapter 1. This leads to repetitive cache misses as explained above in subroutine *cont.*
- Subroutines *quick, cal_u and cal_v*: These subroutines are mainly plagued by *i,j* loop orientation in nested loops. Subroutines *cal_u* and *cal_v* call subroutine *quick* and subroutine *tdma* as discussed above.

3.9 SUMMARY

GHOST is a generalized 2D incompressible structured CFD solver with an ability to perform computations in parallel using an MPI environment. The single node performance of the code was found to be poor (as described in next chapter) due to the high cache miss percentage when it was tested on the in-house clusters. The top six subroutines that contribute to 98% of the run time have been identified and the starting point was to tune these subroutines. The next chapter presents various techniques that improved single node performance as well as speed up across multiple nodes. The single node performance was mainly improved by focusing on reducing cache misses.

CHAPTER-4

4. STAGE ONE PERFORMANCE TUNING RESULTS

This chapter provides a detailed description of stage one of the tuning efforts carried out on the GHOST code. Stage one of the tuning efforts was carried out until December 2004 and was primarily focused on KFC3 and KFC4. Later, techniques that yielded performance improvements on KFC3 and KFC4 were tested on KFC6 architectures. This effort started around September 2008 and fall into stage 2 of the tuning efforts. Although the results on KFC6 architectures fall into stage 2 of tuning effort, in order to comprehend the impact of applying these tuning techniques on older and newer architectures and for comparison purposes, the results on KFC6 architectures are presented as part of stage one along with the ones on KFC3, KFC4. This chapter begins with a description of the types of tests conducted during the tuning effort. The test case is then presented. Later, results from applying some of the tuning techniques discussed in chapter 2 are presented. Changes in the cache behavior of the code are discussed along with the details of performance improvements.

4.1 TYPES OF TESTS

This section describes the kinds of tests that were carried out. They are briefly summarized below:

- **Single Node Performance Tests:** In the current work, the tuning effort is carried out in stages. To assess the impact of changes to GHOST at each stage, walltime, calculated using UNIX functions, is compared between the earlier version and the most tuned version at that point. For assessment purposes, walltime is effectively the average time for a single iteration over a 5000 iteration simulation and it is normalized to eliminate the effect of increasing walltime with increasing grid sizes. These tests are carried out on KFC3, KFC4, KFC5, KFC6I and KFC6A. These commodity clusters have been described in chapter 3. A wide variety of commodity clusters was chosen so as to be able to analyze the effect of optimization techniques on the code running on different architectures. Since the optimizations carried out were not focused around input/output

(I/O) operations, the time taken by I/O operations is not included in the walltime and as such walltime actually reflects the one excluding I/O.

- **Multiple Node Performance Tests:** Speedup, defined as the ratio of walltime on a single node to the walltime on multiple nodes, forms the basis for these kinds of tests. The baseline performance test for CFD codes on commodity clusters is the measurement of multinode speedup, which to the first order is an evaluation of communication time versus computational time as the number of processor nodes used is increased. This is often a critical consideration for CFD simulations on commodity clusters which use relatively inexpensive networks that could create communication delays if the code is not effectively designed. Speedup tests are performed to understand the scalability of a code across multiple nodes.

- **Cache Performance Tests:** These tests are done to explore the connection between cache behavior and the overall performance of the code on various grid sizes. The cache miss rate was determined by the *cachegrind* cache profile simulator, which was described in chapter 3. The focus was on reducing L2 cache misses as they are more expensive than L1 cache misses and a high L2 cache miss rate has a detrimental effect on the performance of a code. As cache-profiling on a code takes considerably more (10 to 100 times longer) time than the original simulation, there was a need to extrapolate from over a series of iterations. Figure 4-1 shows the L2 cache miss rate versus iterations on KFC4 for the original version of GHOST over a series of grid sizes. As observed for the cavity flow test case, there is an initial transient period, but the L2 cache miss rates settle into a near-constant value beyond 400 iterations starting this transition at around 200 iterations for a given grid size. This behavior was observed on all the machines. So, to carry out the cache performance of GHOST, cache profiling was done for 500 iterations on all the machines.

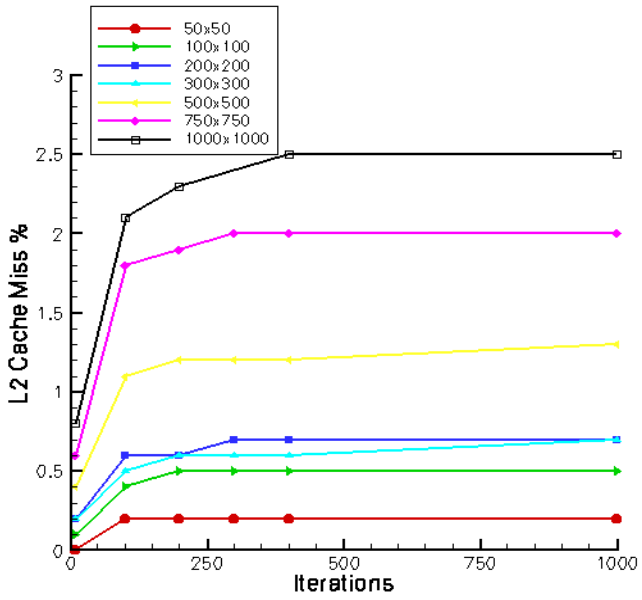


Figure 4-1 L2 cache miss rate for GHOST as a function of iterations from a cold start on KFC4

- **Accuracy Tests:** During the tuning process, although there was no change in underlying algorithm, there was a fair bit of coding change. To confirm that the results from running the simulation have not been altered, tests were run to confirm that the results were in agreement with the solutions obtained by Ghia *et al.* [66] for cavity flow problem.

4.2 TEST CASE

The basic flow test case that has been used to carry out the tuning activity is two-dimensional incompressible driven cavity flow. This is also known as lid-driven cavity flow. This test case was used in proof of concept stage. The cavity is square with a Reynolds number of 1000. This value was chosen to make sure the flow is laminar. This test case has a non-dimensional *u*-velocity value of unity and *v*-velocity value of zero at the top boundary. The walls on the side and the bottom have no-slip boundary conditions. Stationary interior flow is considered as the initial condition. This test case was selected in part because it represents a real and reasonably complex flow from which performance characteristics can be extrapolated to more challenging cases. At the same time, the simple geometry allows for straightforward partitioning and re-gridding, simplifying the evaluation of performance as the grid density and the number of computational nodes is

varied. A schematic diagram of this test case is shown in Figure 4-2. The results from simulations run to completion with original version of the GHOST code are in agreement with Ghia *et al* [66] as shown in Figure 4-3.

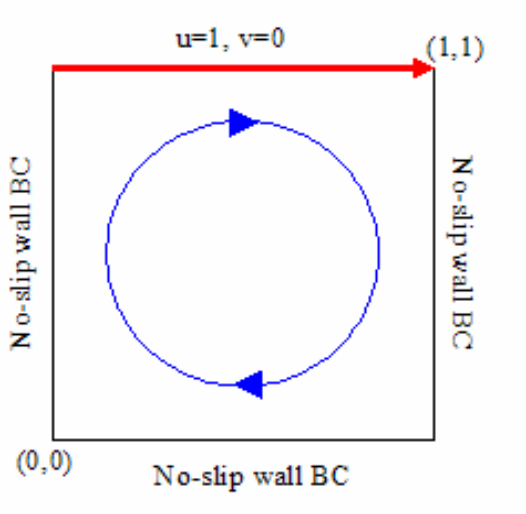


Figure 4-2 Schematic diagram of lid-driven cavity shown with boundary conditions

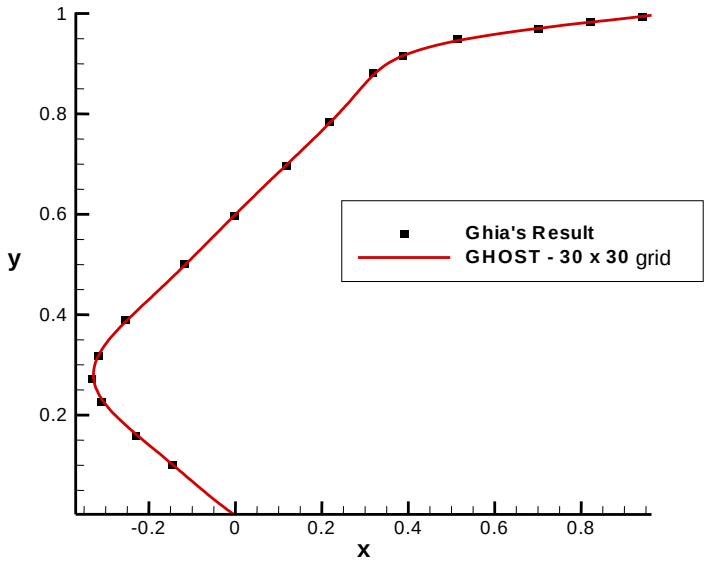


Figure 4-3 The midline *u*-velocity profile for the original version of GHOST

4.3 PERFORMANCE BEHAVIOR OF V0

The initial step of the optimization was to assess the basic picture of performance of the original version (V0) of the GHOST code based on test simulations. While it is

important for a parallel code to scale across multiple nodes, an often ignored fact is that speedup on *n* nodes is calculated based on the performance of a code on a single node. If the code takes exceedingly more time to run on a single node, speedup calculated as ratio of walltime on a single node to walltime on multiple nodes might result in superlinear values. Such superlinear speedup was observed with GHOST, a phenomenon hardly unique in the annals of performance evaluation of parallel codes [68, 69, 70, 71]. The results of speedup tests on KFC3 and KFC4 are presented in Figure 4-4a while the results of speedup tests on KFC6A and KFC6I are presented in Figure 4-4b. For speedup tests, walltime on a single node for various grids of single block ranging 200 x 200 to 1000 x 1000 is calculated. Next, these grids are split into 4, 8, 12 and 16 blocks (external blocking explained in chapter 3) and walltime is calculated by solving them on the same number of nodes as the blocks. Speedup is then calculated as the ratio of the walltime on a single node to walltime on *n* nodes. Ideally, the speedup should be linear. From Figure 4-4a and b, it can be observed that despite the differences in the year of construction (KFC3 in 2003 through KFC6 in 2006) and disparity in hardware (for example KFC6A has an AMD processor while KFC6I has an Intel processor) and networks (relatively fast on KFC3, relatively slow on KFC4), GHOST exhibits dramatically superlinear speedup across a range of problem sizes on all platforms. This behavior is less pronounced on the relatively newer machines KFC6I and KFC6A due to their advanced hardware.

To further examine this dataset, the normalized walltime (walltime normalized by number of grid points and per iteration) is plotted against computational grids of varying sizes. If the code had been running efficiently on a single node, the normalized walltime would have increased as we moved from grids that fit into cache to grids that are considerably larger than cache. After that, the normalized walltime would have essentially remained same even as the size of grid increased. However, this was not what was observed. Normalized walltime for the cavity flow test case over total grid sizes ranging from 30 x 30 to 700 x 700 on five different platforms is shown in Figure 4-5.

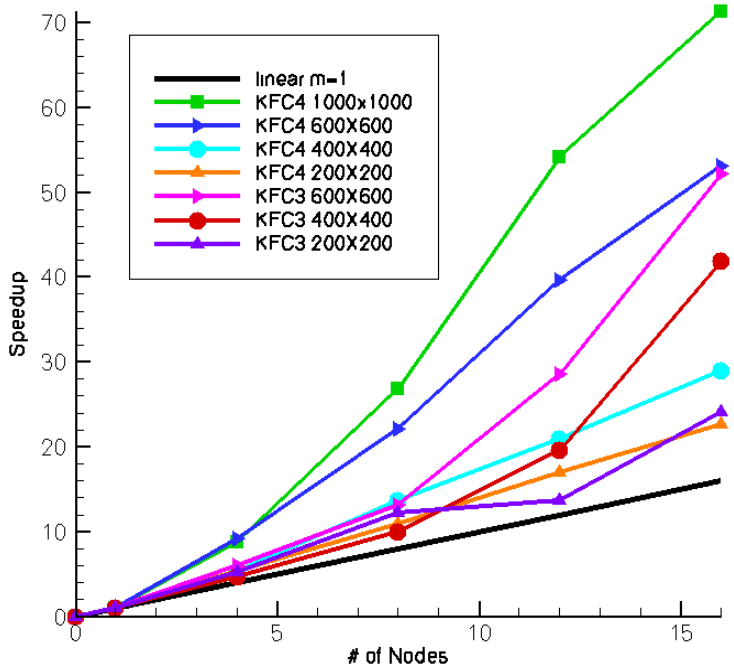


Figure 4-4a Original speedup of GHOST on KFC3, KFC4 for grids of varying size

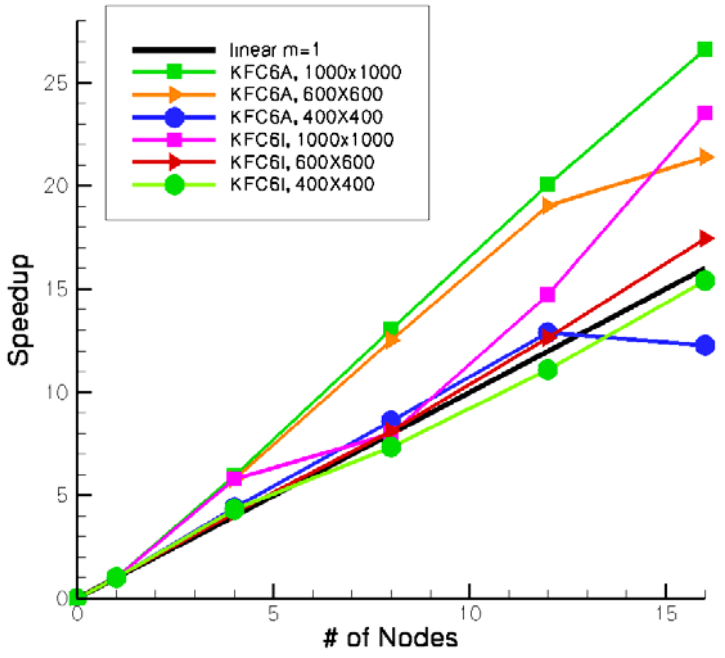


Figure 4-4b Original speedup of GHOST on KFC6A, KFC6I for grids of varying size

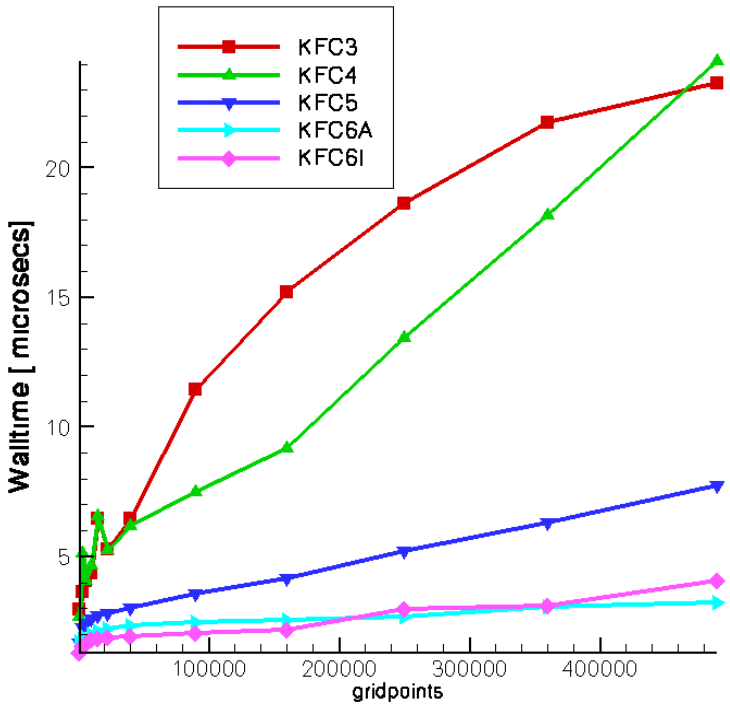


Figure 4-5 Original walltime of GHOST

In the Figure 4-5, walltime is the walltime required to run 5000 timesteps on a dedicated node normalized by the number of grid points. For laminar, steady simulations, the smallest grid, 30 x 30 is effectively contained within L2 cache on all the machines; the largest occupies a majority of the available RAM (512 MB) on KFC3 and KFC4. As suggested by the cavity test problem, all the grids used had an equal number of points in *i* and *j*, a convention that will be used throughout the GHOST analysis. It can be observed that normalized walltime keeps increasing with increasing grid size. This trend is observed irrespective of the architecture of the machine. On older machines KFC3 (2003) and KFC4 (2004), the slope of the line is steeper while in newer machines KFC5 (2005), KFC6A (2006) and KFC6I (2006), the line appears to be flattening out. This is because of faster processors, larger caches (2 MB L2 cache on KFC6A and 1 MB on KFC6I) and a faster Front Side Bus (FSB) that controls the speed of transfer of data from RAM to cache. Increases in walltime with increasing grid sizes can be attributed to high cache misses as evident from Figure 4-6 which represents external blocking (introduced in chapter 3) results on KFC3 for block size of range from 30 x 30 (which effectively fits in

L2 cache) to 70 x 70 along with the results of unblocked grids. The best performance improvement was obtained with the 30 x 30 block grid. As explained above, this is because a 30 x 30 grid can effectively fit into cache. In the case of the 600 x 600 grid the walltime for the 30 x 30 block grid is 1/6 that of the unblocked code on KFC3. This is because fitting the computation problem into cache can largely hide a variety of other program flaws that cause the problem of increasing walltime with increasing grid size as shown in Figures 4-5. In effect, a 30 x 30 external blocking extends the normalized computational speed of the 30 x 30 single grid to much larger grids. The blocked performance is also highly scalable, remaining constant over a wide range of grid sizes. But, in general, the size of the grid on the node appears to be the dominant determinant of the walltime required for the computation in the original version of GHOST.

In order to more deeply explore the connection between cache performance and the above grid dependence, the cache miss rate was measured by the 'cachegrind' cache profile simulator of the Valgrind toolkit. This tool was introduced in chapter 3. As discussed earlier (Figure 4-1), cache miss rate begins to asymptote beyond about 400 iterations for a given grid size. Accepting this asymptotic miss rate as typical for that grid size, a plot of the miss rate versus grid size has been generated. These results are displayed in Figure 4-7 which shows the data cache miss rate (L2D) and overall (data + instruction, L2) miss rate of the L2 cache for GHOST on KFC4 and KFC3. The instruction cache miss rate was negligible for all cases, so the cache variations in performance are dominated by the L2D characteristics. There is also little difference between the Intel Pentium processor on KFC3 and the Athlon processor on KFC4.

In order to explore the correlation between the size of the problem, walltime behavior and the cache miss rate, the memory footprint of the grid was determined by cluster toolkit Warewulf [72]. The memory footprint corresponds to the amount of random-access memory required by the code for a particular grid size. The GHOST code miss rate varies significantly with memory footprint, generally increasing with increasing grid size but with notable spikes at certain points. The direct correlation between the miss rate and the speedup performance of GHOST on KFC4 can be seen in Figure 4-8. The walltime is normalized by the memory footprint to effectively remove the time increase expected due to increased grid size. The resulting curve largely tracks the L2 miss rate for

GHOST. The implication is that much of the superlinear speedup in GHOST is directly traceable to L2 cache performance, as shrinking the grid size will generally improve cache performance.

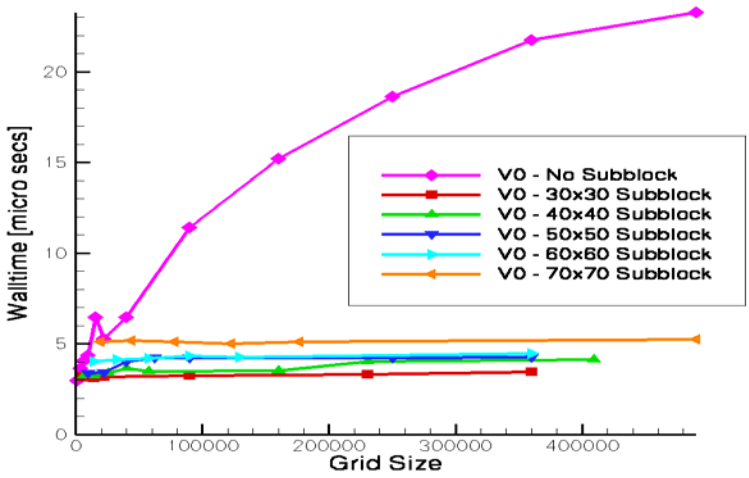


Figure 4-6 Walltime as a function of subgrid size (or grid size for a single node case) for GHOST (V0) on KFC3

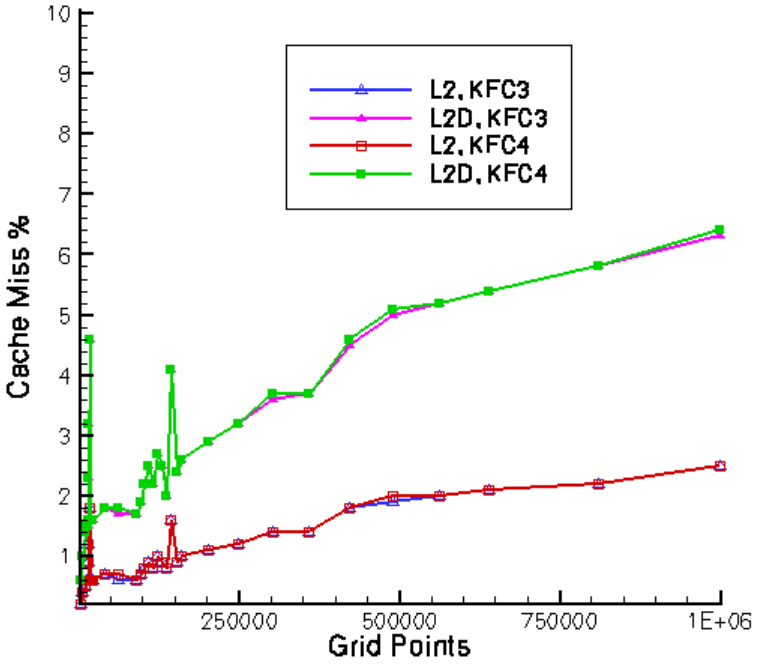


Figure 4-7 L2 and L2D cache miss rate for GHOST (V0) on KFC3 and KFC4

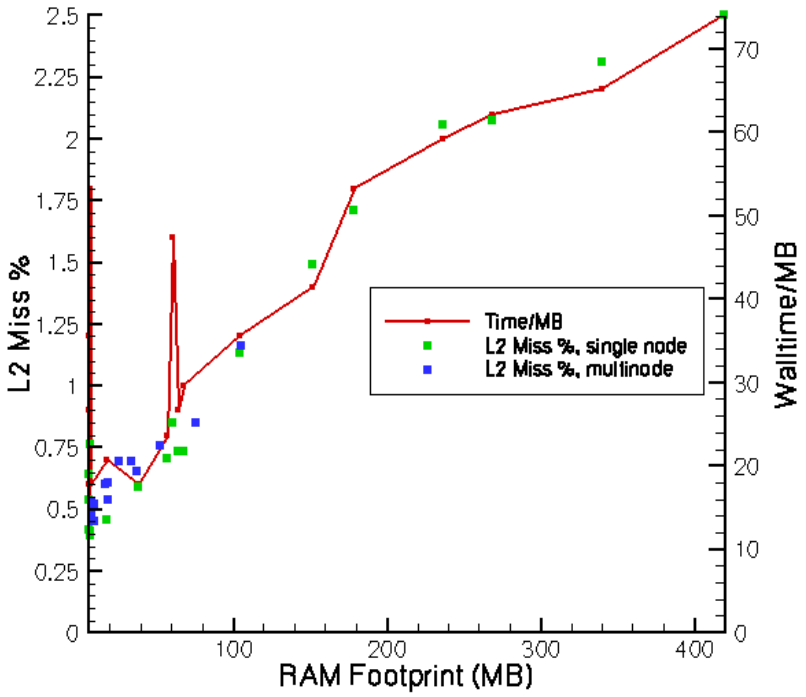


Figure 4-8 Comparisons of L2 cache miss rate on KFC4 (blue and green lines) and the walltime/MB (lines) versus the RAM footprint of the given grid/subgrid for GHOST (V0)

Although external blocking might seem to be a feasible solution to overcome the problem of increasing walltime per grid point, the amount of time it takes to split the grid into cache size blocks puts the programmer at a disadvantage especially when dealing with larger grids. For example, as presented in Figure 4-6, in order to split the grid into cache size blocks, a 600 x 600 grid had to be split into 400 blocks of 30 x 30 each. Although splitting larger grids into smaller blocks that fit into cache yields the best walltime, this process becomes more challenging when dealing with complicated grids of multiple zones and different boundary conditions and thus cannot be used as a standard process to mitigate the problem of increasing walltime with increasing grid sizes. However, the superlinear speedup behavior of GHOST presented above along with the correlation between high cache miss rates and increasing walltime for single zone grids represents an opportunity to tune the code on a single node by improving its cache behavior. This made a strong case for the tuning process to target the L2 cache miss rate

with the aims of reducing the miss rate and attempting to achieve a more uniform distribution.

4.4 TUNING PROCESS – CODE VERSIONS

The original version of the GHOST code is referred to as V0. In order to distinguish various stages of tuning effort, each stage in the tuning process has been associated with a code version *viz.* Version 1 (V1), Version 2 (V2), Version 3 (V3). The following is the list of steps taken during the tuning process.

1. Replacing the allocation/de-allocation scheme with permanent variables.
2. Correcting the orientation of the *i,j* sweeps to the cache-conserving form (*i.e.* outer loop *j*, inner loop *i*) consistent with the storage in memory, (Loop Interchange).
3. Aggressive cleaning of redundant computations, unnecessary divisions, and other excessive mathematical activity.
4. Removal of unwanted if-then structures, particularly on sweeps that do not encompass the full *i,j* grid
5. Restructuring the variables from the single array form, $\phi_1(i,j)$ and $\phi_2(i,j)$, to an array of structures $\Phi(i,j):\phi_1,\phi_2$.

Step (1) proved ineffective resulting in neither a significant change in walltime or cache performance and as such is left out of the subsequent analysis. Steps (2-5) are discussed in detail in later sections of this chapter. Based on the above steps, various versions of the GHOST code were constructed and tested. The following versions are discussed in this chapter.

V0 – original code V1 – original + step 2 V2 – original + steps 2-4
V3- original + steps 2, 5

4.5 CODE CHANGES AND PERFORMANCE TUNING RESULTS

4.5.1 VERSION 1 OR V1

Version-1 or V1 is the result of the first stage of the tuning effort. It was observed that in the original version of code V0, the order (outer loop–*i,* inner loop– *j*) of the majority of the nested *i,j* loops do not allow the compiler to take advantage of the order of data storage in memory. When dealing with large data sets, mismatch in data storage and

data access leads to heavy cache miss rates leading to the poor performance of V0. V1 is the version of GHOST code in which the *Loop Interchange* technique was applied to improve V0. This technique was introduced in chapter 2. The orientation of *i,j* sweeps was corrected to the cache-conserving form (i.e., outer loop *j,* inner loop *i*) to be in consistent with the storage in memory. Subroutines that underwent changes in this stage of tuning are *quick, cal_property, cal_u, cal_v, cont, cal_t* and *cal_tk.*

4.5.1.1 KFC3 and KFC4 Results

The results of applying the *Loop Interchange* technique on KFC3 and KFC4 are presented in Figure 4-9. Despite the disparity in hardware and network, walltime normalized per grid point and per iteration results remain relatively flatter and extend to large grid sizes at least up to one million grid points. For the largest grid (one million grid points), walltime for V1 on KFC3 is 25% that of V0 while on KFC4 performance gains are much more pronounced; for larger grids, V1 is at least 6 times faster than V0.

As shown in Figure 4-10, the observed performance gains can be attributed to reduction in cache misses in V1 when compared to V0. For example, for V1 on KFC4, D1 cache miss (data calls that miss L1 cache are called D1 cache misses) rates vary from 0.3 (for smaller grids) to 0.5 (for larger grids) of those for V0. L2D and L2 cache miss rates settle to a near constant value of 1.7 and 0.6 respectively through out the problem size range while these values increased with increasing grid size for V0.

Figure 4-11a and b plot walltime and the number of data calls, D1 cache misses, and L2D cache misses normalized by number of grid points versus grid size for V0 and V1 on KFC4. Note the cache data is taken from Valgrind simulations ranging from 400 to 600 iterations and extrapolated to 5000 iterations based on the arguments previously presented in conjunction with Figure 4-1. As seen from Figure 4-11b, walltime improvements can be directly traced to decrease in D1 cache misses and L2D cache misses when compared to those in V0 (Figure 4-11a). L2D misses largely trace the walltime plot in V1 on KFC4 as in V0; the difference being L2D misses normalized by grid points flatten out at 200000 grid points in V1 while they tend to keep increasing in V0. Also, for grid sizes beyond 200x200, V1 at least has 50% fewer D1 cache misses than V0 thus explaining drastic improvements in walltime beyond this grid size. Absolute

walltime values for V0 and V1 for various grid sizes on KFC4 are shown in Table 4-1 for comparison.

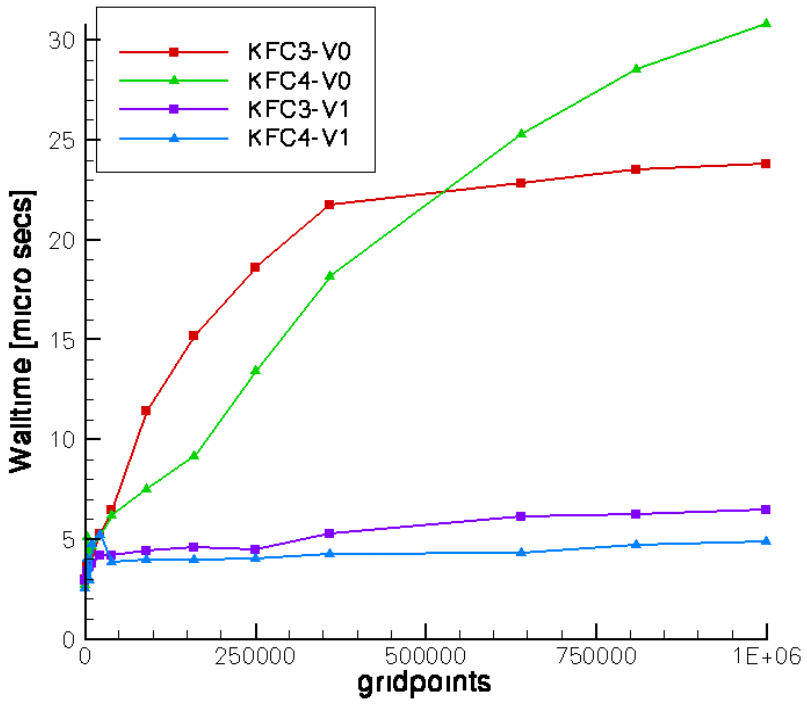


Figure 4-9 Comparison of walltime between V0 and V1 on KFC3 and KFC4

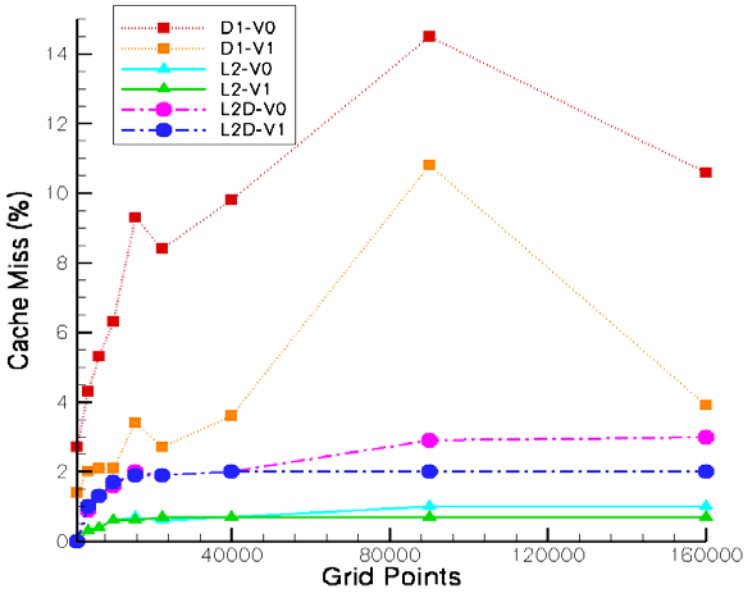
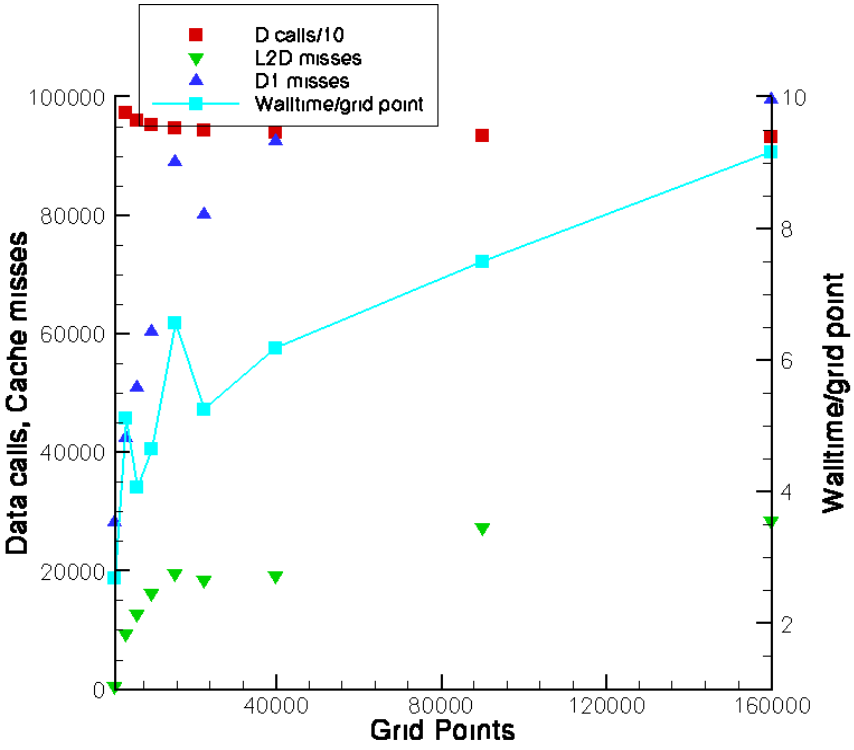
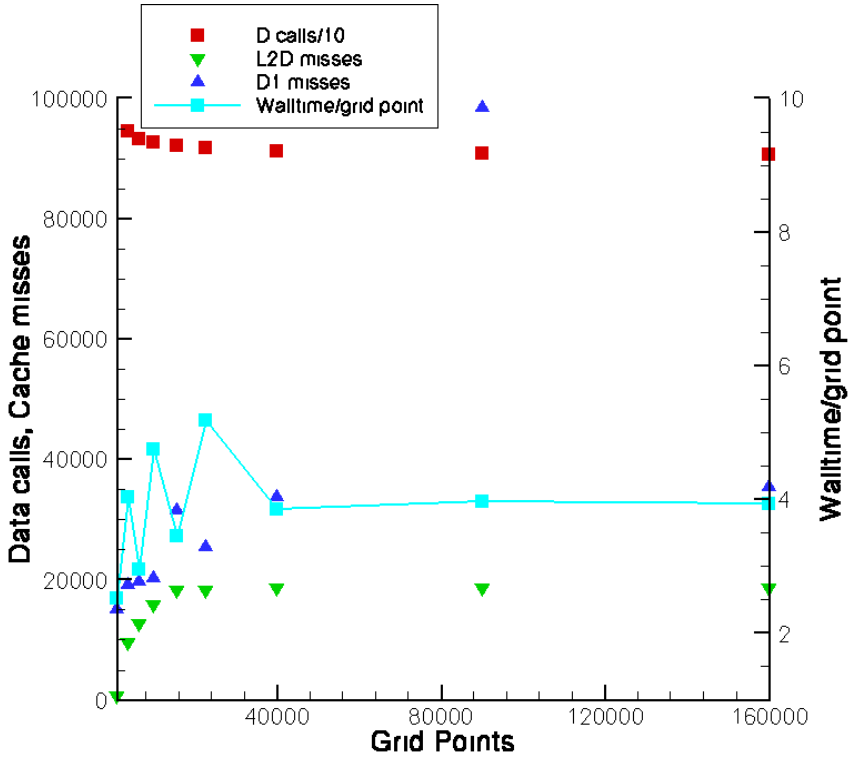


Figure 4-10 Comparison of D1, L2 and L2D cache miss rates for V0 and V1 on KFC4



(a)



(b)

Figure 4-11 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC4 for GHOST (a) Version 0 (b) Version 1

Table 4-1 Comparison of Walltime (in seconds) for V0 and V1 on KFC4

Grid size	V0-Walltime	V1-Walltime	% improvement
30 x 30	12.09	11.37	5.96
60 x 60	91.97	72.77	20.88
80 x 80	130.47	94.59	27.50
100 x 100	232.67	232.43	0.10
200 x 200	1235.4	769.07	37.75
300 x 300	3371.63	1783.43	47.10
400 x 400	7324.08	3152.79	56.95
500 x 500	16770.23	5028.45	70.02
600 x 600	32662.18	7615.41	76.68
700 x 700	59122.85	12698.62	78.52
800 x 800	80974.31	13845.95	82.90
900 x 900	115577.1	18958.21	83.60
1000 x 1000	154196	24295.93	84.24

4.5.1.2 KFC6 Results

Results of performance gains on KFC6A and KFC6I are presented in Figure 4-12. Performance gains are relatively less on these clusters when compared to the older and slower machines KFC3 and KFC4. This is not unexpected and can be attributed to faster processors and bigger caches along with a faster Front Side Bus (FSB) as discussed earlier. For the largest grid (one million grid points), walltime for V1 on KFC6A is 27% that for V0 while on KFC6I performance gains are higher; for the largest grid, V1 is almost twice as fast as V0. Improvements in walltime can again be attributed to improvements in cache behavior with the exception of these improvements mainly coming from D1 cache. As presented in Figure 4-13, while L2 and L2D cache miss rates remain similar, D1 cache misses reduce by 50% through out the problem size range in V1. Thus, walltime improvements can largely be attributed to improvements in D1 cache behavior. This is evident from the Figures 4-14a and b. For comparison purpose, normalized walltime and the number of data calls (sum of memory reads and memory writes) normalized by grid points divided by 10, D1 cache misses and L2D cache misses

normalized by grid points are presented in Figures 4-14a and b. Note that the cache data is taken from Valgrind simulations for 500 iterations and extrapolated to 5000 iterations based on the arguments previously presented in conjunction with Figure 4-1. D1 misses for V1 are at least 50% less than those in V0.

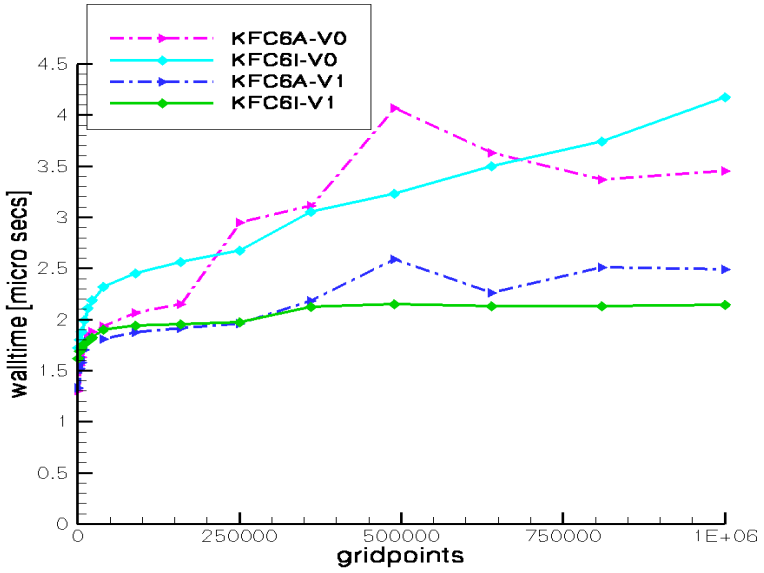


Figure 4-12 Comparison of walltime between V0 and V1 on KFC6A and KFC6I

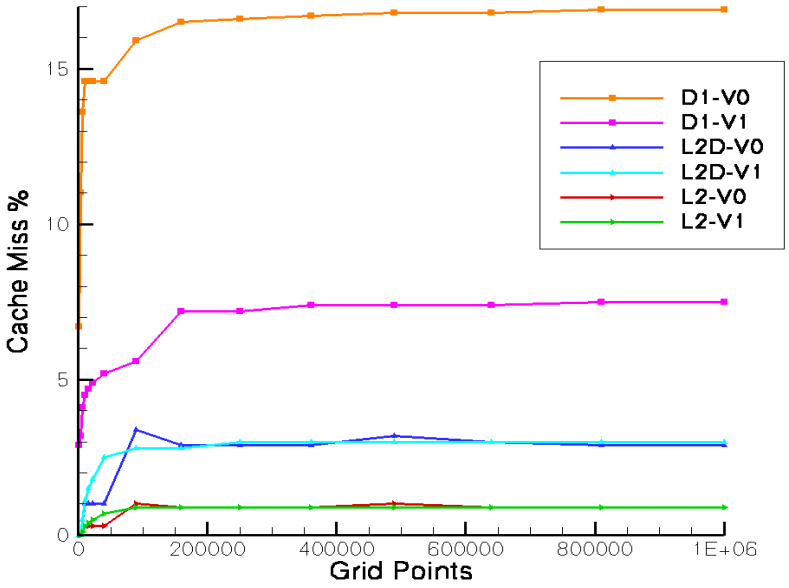
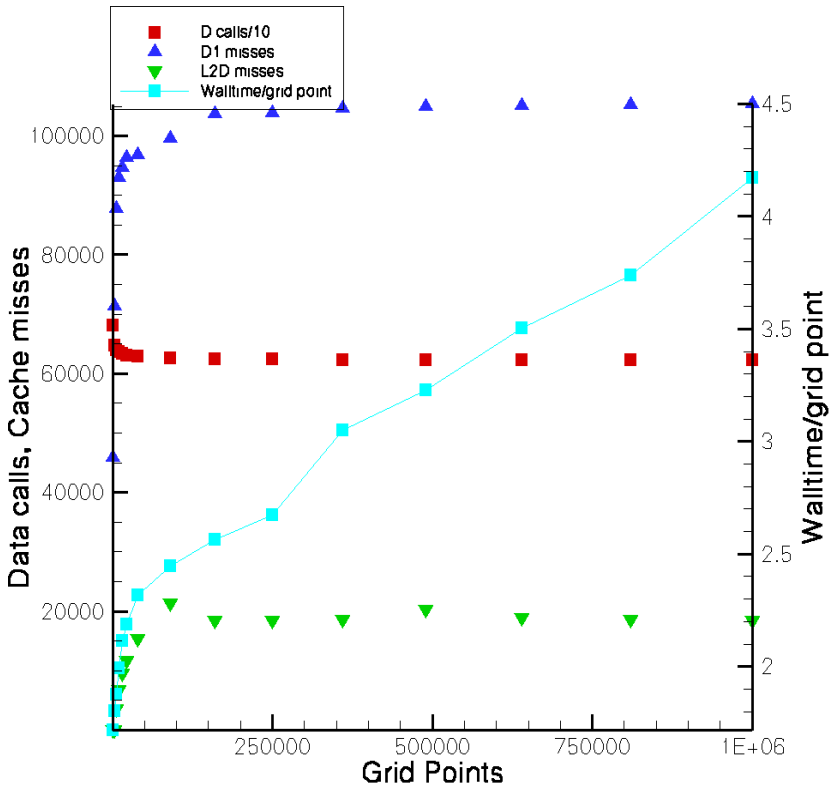
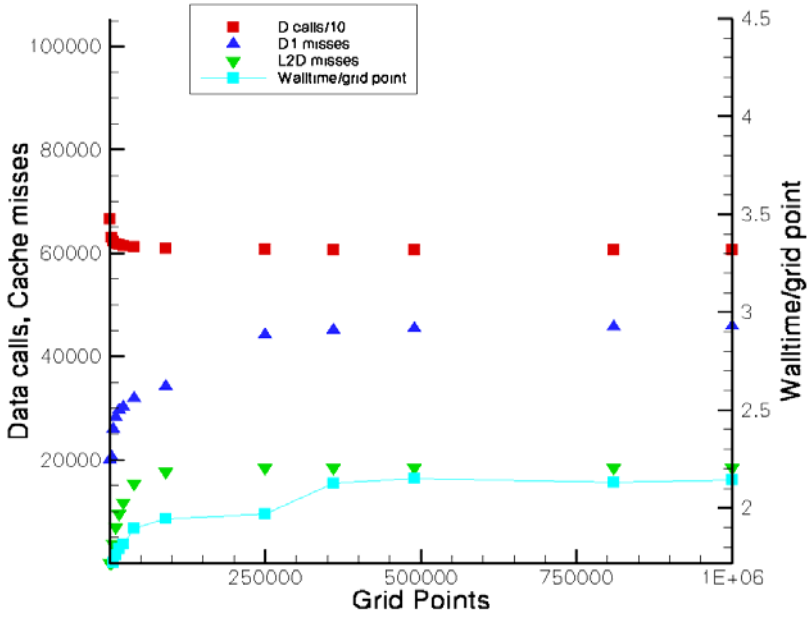


Figure 4-13 Comparison of D1, L2 and L2D cache miss rates for V0 and V1 on KFC6I



(a)



(b)

Figure 4-14 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC6I for GHOST (a) V0 (b) V1

4.5.2 VERSION 2 OR V2

Version 2 or V2 is the result of the second stage of the tuning effort. This version is an improved version of V1. In other words, V2 is V0 with *Loop Interchange* technique implemented (V1) along with the techniques implemented to avoid redundant operations. The focus of this tuning stage was to avoid unnecessary mathematical computations in the code and to speed up at least some of the mathematical calculations by incorporating changes without modifying the underlying algorithm. These are described in the next few paragraphs.

4.5.2.1 Avoiding Unnecessary Recalculations

When iterating inside a loop, using pre-calculated values wherever possible instead of recalculating them every time saves considerable amount of time especially if similar values are being calculated repeatedly. This technique was briefly discussed in chapter 2 under 'Optimizing Floating Point Operations' section.

4.5.2.2 Avoiding Division inside loops

In V1 (as well as V0), there are a few subroutines (*cont, cal_u, cal_v)* that do repeated divisions inside nested loops. The idea is to calculate the reciprocal value, store and later reuse the result without having to repeat the calculation, a technique often termed as implementing *Reciprocal Cache* [42, 43]. In V1, as the subroutine *cont* is the most expensive of all subroutines, in V2 *Reciprocal Cache* was implemented for this subroutine only. An example is shown in Figure 4-15. In subroutine *cont,* there were 52 instances where repeated division operations were replaced by a pre-calculated value that was calculated outside nested loop. Repeated divisions (52 in total) in V1 were being done inside 4 nested *i, j* loops that span across the dimensions of the grid. As subroutine *cont* is executed in every iteration, repeated divisions were being done for every iteration. For example in a 600 x 600 grid, this means in a given iteration, 52*600*600 divisions were being done in subroutine *cont.* With high cost of division operations (details discussed in chapter 2), this was a bottleneck for optimum performance of this subroutine in V1 (and V0). In V2, the reciprocal values calculated only once at the beginning of the subroutine *cont* are used throughout its code avoiding further division operations. This essentially means 52 repeated division operations per iteration have been replaced by a single division operation per iteration. This reduction in number of division operations

leads to savings in machine cycles. For example, cycle time for a division operation on AMD 64-bit architecture is 71 cycles while on Intel 64-bit it is 161 cycles [30] and because of this variation in cost of division operations between architectures, actual gains depend on architecture of a machine on which *Reciprocal Cache* is being implemented.

4.5.2.3 Merging Loops

In V1, two adjacent nested loops (shown in Figure 4-16) were found in subroutine *cal_u*. A simple analysis confirmed that the algorithm does not alter when these two loops are merged.

Although with this change, two new loops are introduced due to variation in the upper bounds of the merged loop, speed up was still observed as this reduces the number of *i,j* sweeps due to loop merging. This technique encourages *temporal locality* as the number of iterations that separate successive accesses to a given reused data is reduced if same data elements were being referenced in two consecutive loops before their merge.

Subroutine *cont* in V1:

```
   DO j = 1, nj
        DO i = 2, ni
             ……….

              auu = 1. / au (i, j) + rp * (1./au(i+1, j)-1./au(i, j))
              avv = 1. / av (i, j) + rp * (1./av(i+1, j)-1./av(i, j))
             ……….

        END DO
   END DO
```

Subroutine *cont* in V2:

```
REAL (high), ALLOCATABLE, DIMENSION (:, :) :: inv_of_au, inv_of_av
!memory allocation
ALLOCATE (inv_of_au(ni,nj)
ALLOCATE (inv_of_av(ni,nj)
     inv_of_au = 1./au
      inv_of_av = 1./av
      DO j = 1, nj
       DO i = 2, ni
           ……….
            auu = inv_of_au (i,j) + rp * (inv_of_au(i+1, j) - inv_of_au(i,j))
            avv = inv_of_av (i,j) + rp * (inv_of_av(i+1,j) - inv_of_av(i,j))
         ……….
       END DO
     END DO
```

Figure 4-15 Using Reciprocal Cache in subroutine *cont*

For example, in V1 for a 600 x 600 grid and a given set of values of *i* and *j*, the same values of *u(i,j)* and *du(i,j)* would be accessed 360000 sweeps later because of an immediate nested loop following the first loop. As this number is high, the compiler does not expect the requirement for these values soon. When this happens, because of the limited size of the cache memory, the compiler does not tend to retain these values in cache. This increases the number of memory references.

On the other hand, in the case of V2, for the same 600 x 600 grid, as the same value of *u(i,j)* is being used in two different operations in a single nested loop, there is a possibility of improvement of *temporal locality*. The two superfluous reloads in V1 have been avoided in V2 as the loops are merged into a single one as shown in Figure 4-16.

Subroutine *cal_u* in V1:

```
      DO j = 2, njm1
        DO i = 2, nim1
         u (i, j) = u (i, j) + du (i, j)
        END DO
      END DO
      DO j = 1, nj
        DO i = 1, ni
         upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
        END DO
      END DO
```

Subroutine *cal_u* in V2 (unnecessary loop merged with the first one):

```
      DO j = 2, njm1
       DO i = 2, nim1
        u (i, j) = u (i, j) + du (i, j)
        upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
       END DO
      END DO
 DO i=1, ni
  j=1
    upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
  j=nj
    upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
 END DO
 DO j=2, nj-1
  i=1
    upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
  i=ni
    upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
 END DO
```

Figure 4-16 Merging nested loops in *cal_u*

For a given set of values of *i* and *j*, the cache controller will load the element *u(i,j)* only once. The attempt is also made to hold on to this data element as this value is being

reused in the next operation. This leads to fewer memory references and improves *temporal locality.* Similar changes have been implemented in the same subroutine (*cal_u)* at a different place as shown in Figure 4-17a and in subroutine *cal_v* as shown in Figures 4-17b.

4.5.2.4 Getting rid of IF-THEN in loops

As described in chapter 2, conditional statements inside nested loops are expensive. In V1 (subroutine *cont)*, the IF-THEN-ELSE structure was found inside nested loops. The nested loop was modified by removing the IF-THEN-ELSE structure out of the loop. This is presented in Figure 4-18.

The loop merging technique was then applied to this part of subroutine *cont*. In order to deal with differences in the bounds of two nested loops (shown in Figure 4-18), sweeps along *i* and *j* have been added in V2. As the value of *i* ranges from 0 to *ni*, the IF condition would be true in 4 cases, when i=0 or 1 or when i=*nim1* (i.e., *ni*-1) or when i=*ni*. So as to be able to merge this loop with the other loop in the same subroutine, these 4 cases have been handled outside the original loop. This technique is a different way of implementing *Inlining*, a technique described in chapter 2.

4.5.2.5 KFC3 and KFC4 Results

The results of applying the above discussed techniques on KFC3 and KFC4 are presented in Figure 4-19. We were not able to collect walltime information for large problem sizes beyond 800 x 800 (640000 grid points) on KFC4 due to problems with stack; but, it is believed that the data collected is sufficient enough to realize the benefits of V2 over V1 because the largest problem size (700 x 700) for which walltime has been recorded is much larger than L2 cache. Walltime for V2 is better than V1 by 5-10% for most of the problem sizes on both clusters. Table 4-2 presents absolute walltime values for V1 and V2 on KFC4. Performance gains are not drastic as only one subroutine (*cont)* was tuned in this stage. Later, techniques that have been implemented in subroutine *cont* have been applied to remaining part of the code. These changes are presented in the next chapter as this fell under a separate stage of tuning GHOST.

Figure 4-20 presents cache miss rates for V1 and V2 on KFC4 for grids only up to 160000 grid points due to stack problems encountered while running Valgrind beyond

this grid size. Cache behavior in V2 does not show much improvement when compared to V1; in fact, the D1 miss rate for V2 is more than that for V1. This is attributed to the fact that tuning effort in this stage was focused on reducing the number of data calls (as discussed above) and this value being in the denominator (while calculating D1 miss rate) might lead to a larger fraction (D1 miss rate). This is evident from the Figures 4-21a and b. For comparison purposes, normalized walltime and the number of data calls (sum of memory reads and memory writes) normalized by grid points divided by 10, D1 cache misses and L2D cache misses normalized by grid points are presented in Figures 4-21a and b.

Note that the cache data is taken from Valgrind simulations for 500 iterations and extrapolated to 5000 iterations based on the arguments previously presented in conjunction with Figure 4-1. From Figures 4-21a and b, D1 calls in V2 are at least 3% less than those in V1 for grid sizes presented. It can also be observed that L2d misses largely track walltime plot across all grid sizes. From this it is evident that even though we are not able to explain walltime gains with improvements in cache miss rates for V2 (in Figure 4-20), combination of all three types (Data calls, L2d and D1 misses) of cache activity provides an explanation of walltime behavior confirming the effect of cache efficiency on code performance.

4.5.2.6 KFC6 Results

Results of performance gains for V2 on KFC6A and KFC6I are presented in Figure 4-22. Again, due to faster processors and bigger caches, larger walltime improvements of only 10% are observed. Even on KFC6I, walltime improvements in V2 do not correspond to improvements in cache miss rates. Cache miss rates for V1 and V2 are presented in Figure 4-23 for comparison. However, as discussed earlier, the focus of the tuning effort in this stage was to avoid unnecessary and repeated calculations inside loops in order to reduce the number of memory references. As expected, V2 is characterized by fewer (by 5%) data references than V1 on KFC6I as shown in Figures 4-24a and b and this is the probable reason for improvement in walltime in V2.

Subroutine *cal_u* in V1:

```
      DO j = 1, nj
        DO i = 1, ni
      ……….
      END DO
    END DO
      DO j = 1, nj
        DO i = 1, ni
        ap (i, j) = ae (i, j) + aw (i, j) + an (i, j) + as (i, j) + ap (i, j)
        IF (vol(i, j) < 1.e-20 .OR. inx(i, j) == 1) THEN
          ap (i, j) = great
          ae (i, j) = 0.
          aw (i, j) = 0.
          an (i, j) = 0.
          as (i, j) = 0.
          rhs (i, j) = 0.
        END IF
        au (i, j) = ap (i, j)
        ap (i, j) = ap (i, j) / urfu
      END DO
    END DO
```

Subroutine *cal_u* in V2 (unnecessary loop megred with the first one):

```
  DO j = 1, nj
   DO i = 1, ni
      ……….
  !! mixing loops
  ap (i, j) = ae (i, j) + aw (i, j) + an (i, j) + as (i, j) + ap (i, j)
         IF (vol(i, j) < 1.e-20 .OR. inx(i, j) == 1) THEN
           ap (i, j) = great
           ae (i, j) = 0.
           aw (i, j) = 0.
           an (i, j) = 0.
           as (i, j) = 0.
           rhs (i, j) = 0.
         END IF
         au (i, j) = ap (i, j)
         ap (i, j) = ap (i, j) / urfu
!! end of mixing do loops
       END DO
     END DO
```

Figure 4-17a Merging nested loops in *cal_u*

Subroutine *cal_v* in V1:

```
        DO j = 1, nj
          DO i = 1, ni
          ……….
        END DO
      END DO
        DO j = 1, nj
         DO i = 1, ni
          ap (i, j) = ae (i, j) + aw (i, j) + an (i, j) + as (i, j) + ap
(i, j)
          IF (vol(i, j) < 1.e-20 .OR. inx(i, j) == 1) THEN
            ap (i, j) = great
            ae (i, j) = 0.
            aw (i, j) = 0.
            an (i, j) = 0.
            as (i, j) = 0.
            rhs (i, j) = 0.
          END IF
          av (i, j) = ap (i, j)
          ap (i, j) = ap (i, j) / urfv
        END DO
      END DO
```

Subroutine *cal_v* in V2 (unnecessary loop merged with the first one) :

```
       DO j = 1, nj
          DO i = 1, ni
          ……….
          !! mixing do loops
 ap (i, j) = ae (i, j) + aw (i, j) + an (i, j) + as (i, j) + ap (i, j)
          IF (vol(i, j) < 1.e-20 .OR. inx(i, j) == 1) THEN
            ap (i, j) = great
            ae (i, j) = 0.
            aw (i, j) = 0.
            an (i, j) = 0.
            as (i, j) = 0.
            rhs (i, j) = 0.
          END IF
          av (i, j) = ap (i, j)
          ap (i, j) = ap (i, j) / urfv
!! end of mixing do loops
        END DO
      END DO
```

Figure 4-17b Merging nested loops in *cal_v*

Table 4-2 Comparison of Walltime (in seconds) for V1 and V2 on KFC4

Grid size	V1-Walltime	V2-walltime	% improvement
30x30	11.37	11.58	-1.84
60x60	72.77	48.98	32.69
80x80	94.59	93.62	1.02
100x100	232.43	155.72	33.00
200x200	769.07	717.83	6.66
300x300	1783.43	1576.24	11.61
400x400	3152.79	2936.26	6.86
500x500	5028.45	4736.61	5.80
600x600	7615.41	7131.86	6.34
700x700	12698.62	11584.7	8.77
800x800	13845.95	13619.22	1.63

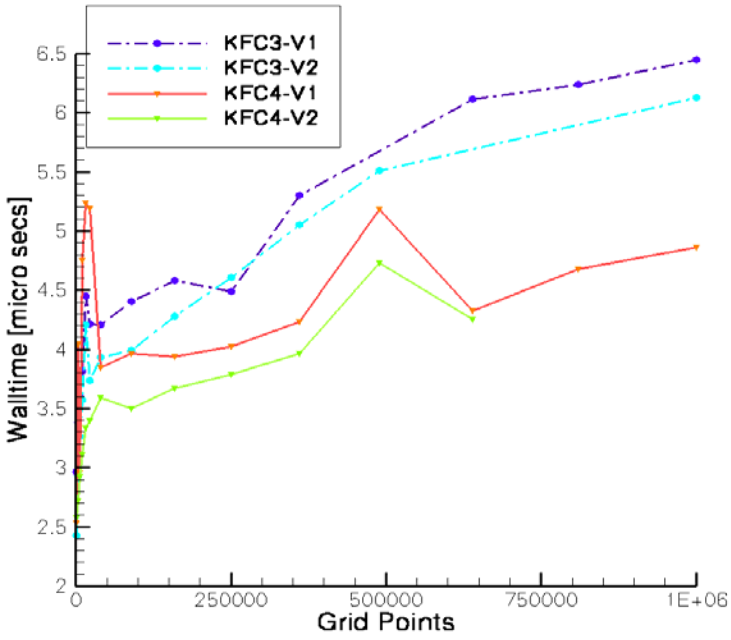


Figure 4-19 Comparison of walltime on KFC3 and KFC4 for V2 and V1

Subroutine *cont* in V1:

```
      DO j = 1, nj
         DO i = 0, ni
           IF (i <= 1 .OR. i >= nim1) THEN
               ……….
           ELSE
               ……….
           END IF
         END DO
       END DO
         DO j = 0, nj
             DO i = 1, ni
           IF (j <= 1 .OR. j >= njm1) THEN
               ……….
           ELSE
               ……….
           END IF
         END DO
       END DO
```

Subroutine *cont* in V2:

```
! additional lines due to changes in "do" loop starting
      i = 0
      DO j = 1, nj
        ……..
      END DO
      i = 1
      DO j=1, nj
        ……..
       END DO
       i = nim1
      DO j=1, nj
        ……..
       END DO
       i = ni
      DO j=1, nj
       ……..
       END DO
! end of additional lines due to change in "do" loop
      DO j = 1, nj
       DO i = 2, nim2
! IF CONDITION REMOVED
          ……..
        END DO
      END DO

! second set of additional lines begin
 j=0
 DO i = 1, ni
 ……..
END DO
 j = 1
 DO i = 1, ni
……..
END DO
 j = njm1
 DO i = 1, ni
……..
END DO
 j = nj
 DO i = 1, ni
……..
END DO
! end of second set of additional lines
        DO j = 2, njm2
            DO i = 1, ni
! IF CONDITION REMOVED
……
        END DO
      END DO
```

Figure 4-18 Removing IF-THEN-ELSE inside loops in V1

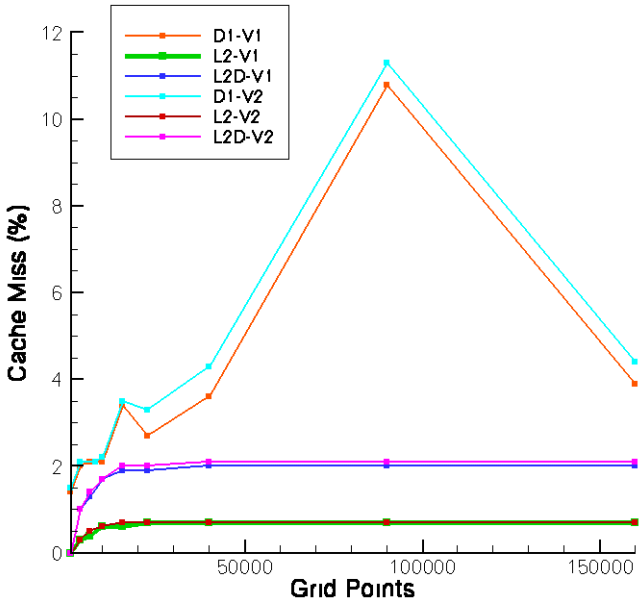
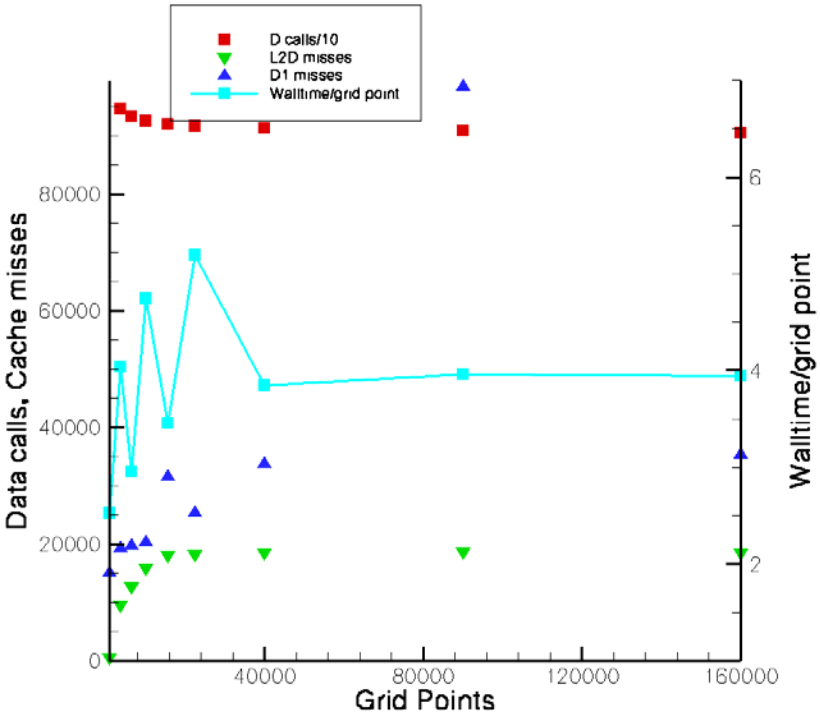
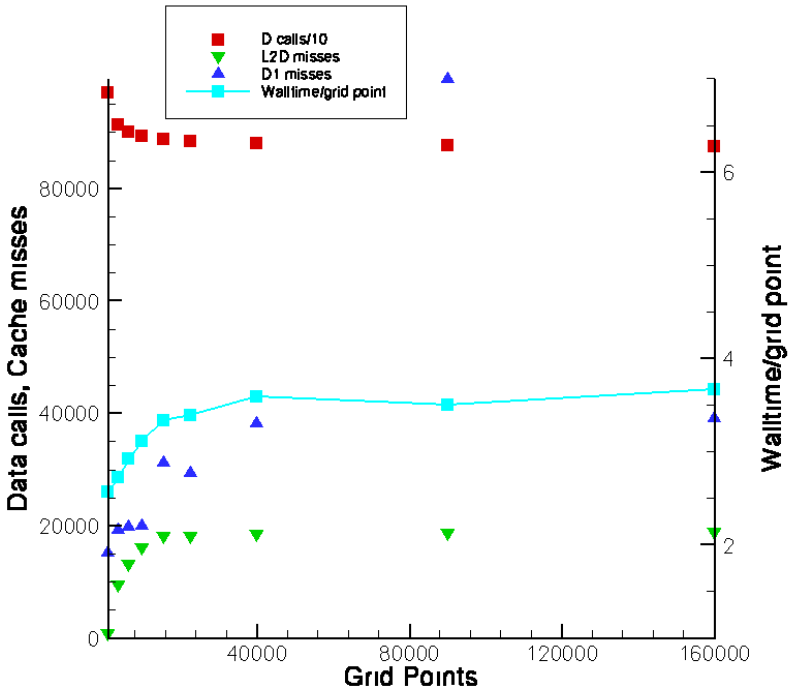


Figure 4-20 Comparison of D1, L2 and L2D cache miss rates for V1 and V2 on KFC4



(a)



(b)

Figure 4-21 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC4 for GHOST (a) V1 (b) V2.

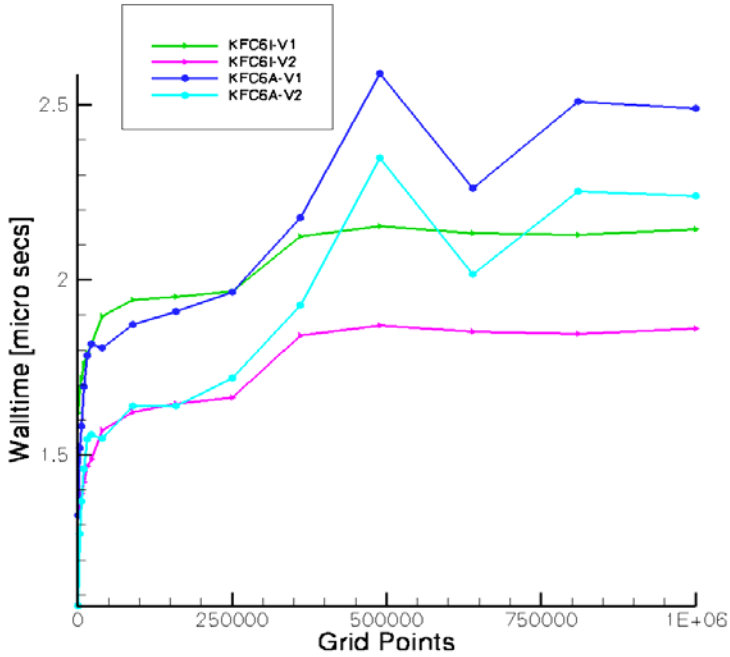


Figure 4-22 Comparison of walltime between V1 and V2 on KFC6A and KFC6I

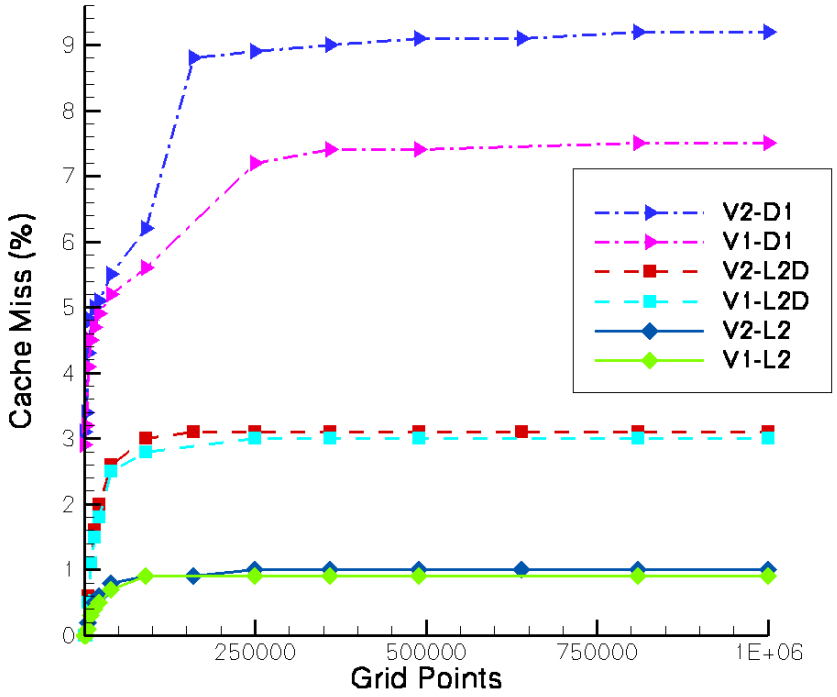
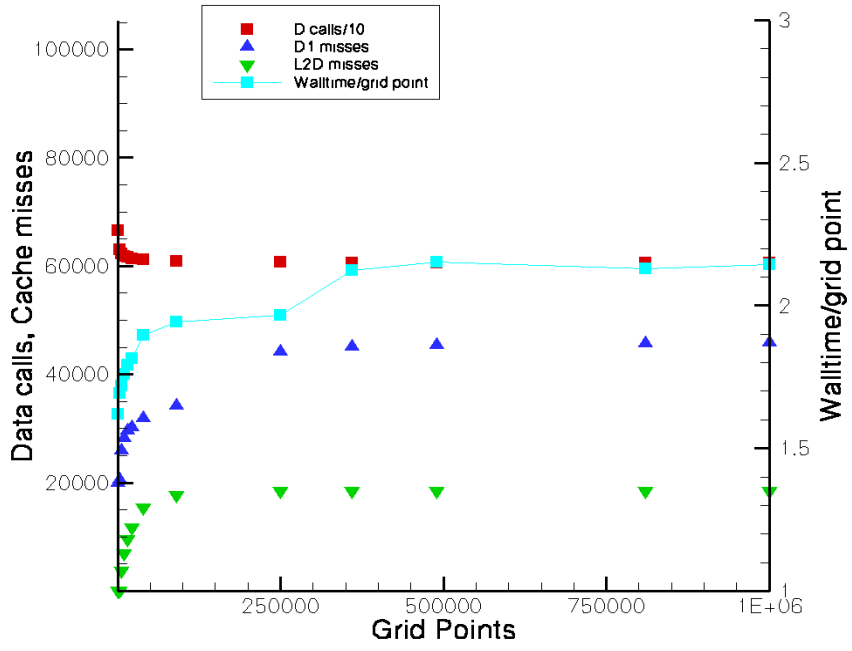
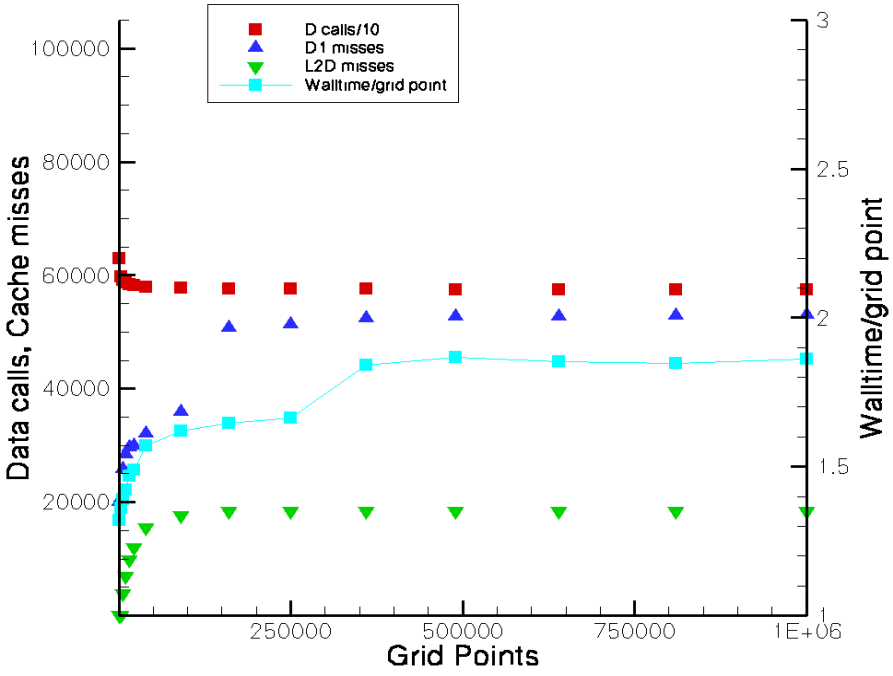


Figure 4-23 Comparison of D1, L2 and L2D cache miss rates for V1 and V2 on KFC6I



(a)



(b)

Figure 4-24 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC6I for GHOST (a) V1 (b) V2

4.5.3 VERSION 3 OR V3

V3 is the result of the third stage of the tuning effort. In order to understand the effect of usage of data structures, the changes done to the code from V1 to V2 have not been implemented in V3. Thus, V3 is V1 with data structures implemented in place of arrays to aid in data fetch. The focus of this tuning stage was to avoid data misses that might be possible due to usage of arrays in the code as explained in chapter 2. The subroutines that have undergone modification with implementing structures are *cal_u, cal_v, cont, quick* and *tdma.*

As the current work focused on optimizing the laminar part of the code, there was a need for preserving the part of the code that deals with turbulent flows. Subroutines *quick* and *tdma* were being used for both laminar and turbulent flow type problems. Modifying these subroutines would have essentially modified a part of the code that deals with turbulent flows and so there was a need for two new subroutines customized for laminar flows. They are *quick_struct* and *tdma_struct.* These two subroutines essentially

have the same code as subroutines *quick* and *tdma* respectively except that they have arrays of data structures implemented in them rather than arrays. In V3, all function calls to *quick* and *tdma* have been replaced by *quick_struct* and *tdma_struct* respectively and the part of the code that deals with turbulent flows is left in its original state.

Subroutines *cal_u, cal_v, cont, quick* and *tdma* have several lines of code that perform arithmetic calculations on elements of various arrays at a grid point or its nearby points. An example of such a calculation is:

```
ap(i,j) = ae(i,j) + aw(i,j) + an(i,j) + as(i,j)
```

Operations like this need the compiler to fetch the data from multiple arrays. Such operations involve arithmetic calculations on the data from the same grid point or surrounding grid points. This makes a strong case for using data structures in place of arrays because when such variables are declared inside a data structure and when the code is modified to use an array of such data structures, the compiler can fetch values of the required variables at once without the possibility of data cache misses. This is because when the processor requests a set of variables, a cache line is loaded into cache and because arrays are replaced by data structures, all variables that are required in arithmetic operations are found in cache leading to fewer data misses. In GHOST, in a given arithmetic operation, the possibility of needing to access values beyond a cache line does not arise because arithmetic operations are performed on variables at a given grid point or its immediate neighboring points. This feature of GHOST makes a strong case for using data structures because arithmetic operations that do not involve data elements located far away from each other on a grid benefit from usage of data structures because of the above explained concept. Details of using data structures in place of arrays are presented in Figure 4-25.

4.5.3.1 KFC3 and KFC4 Results

The results of incorporating arrays of data structures in place of arrays on KFC3 and KFC4 are presented in Figures 4-26a and b. This figure presents comparisons between V1 and V3 because as discussed earlier, V3 is a modified version of V1 (leaving behind all the changes that were incorporated in V2). Performance gains are up to 40% on KFC3 while the gains are lower on KFC4.

One distinct feature of V3 is that the normalized walltime values on KFC3 remains practically same after reaching 125 x 125 grid while for V1 on KFC3, walltime values kept increasing with increasing grid size beyond 125 x 125 grid. Table 4-3 presents normalized walltime values for V1 and V3 on KFC3 for comparison. On KFC4, techniques implemented in V3 remove the sporadic behavior (150 x 150 and 700 x 700 grids) in V2 and the smooth behavior of the walltime plot is obvious from the zoomed plot in Figure 4-26b. Performance gains are more pronounced for larger (250000) grids and gains up to 20% (when compared to V1) are realized in V3 on KFC4.

Valgrind results on KFC4 are presented in Figure 4-27. Drastic reduction in D1 cache misses are seen for grid 300 x 300 on KFC4 although this does not translate to huge gains in walltime. Cache miss rates between V1 and V3 do not differ by large numbers. However fewer D1 misses (by as much as 50%) as shown in Figures 4-28a and b contribute to improvements in walltime.

Data declaration in V1/V2:

```
REAL (high), DIMENSION (:, :) :: ae, aw, an, as, ap, rhs
```

Data declaration in V3:

```
TYPE struct_aewnsp
      REAL (high) :: ae, aw, an, as, ap, rhs
     END TYPE struct_aewnsp
TYPE (struct_aewnsp), POINTER, DIMENSION (:,:) :: aewnsp
```

Example of array usage in V1/V2:

```
DO j = 1, nj
  DO i = 1, ni
    ……
     ap(i,j)=ae(i, j)+aw(i,j)+an(i,j)+as(i,j)+ap(i, j)
     ……
  END DO
END DO
```

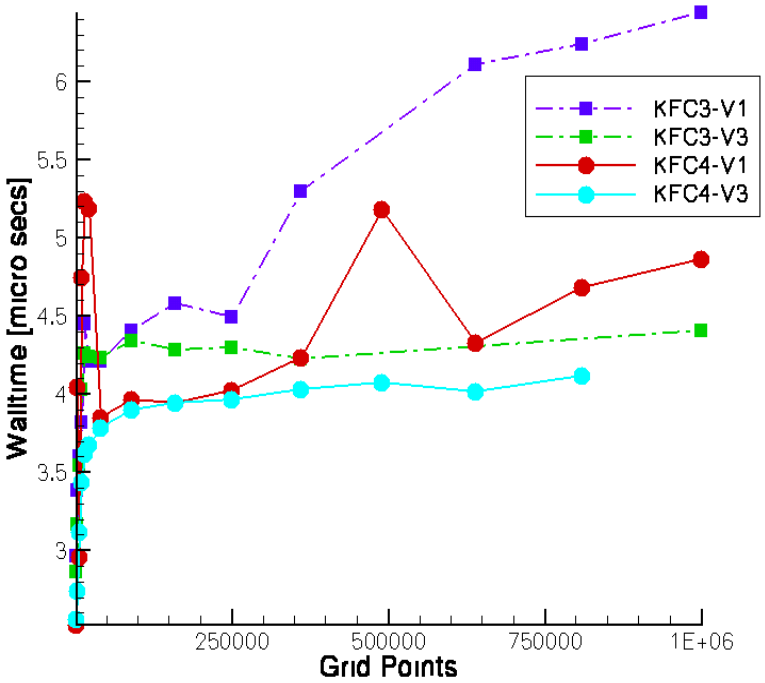
Example of Structure Usage in V3:

```
TYPE (struct_aewnsp), POINTER, DIMENSION (:,:) :; aewnsp
DO j = 1, nj
  DO i = 1, ni
    ……
aewnsp(i,j)%ap = aewnsp(i,j)%ae + aewnsp(i,j)%aw + aewnsp(i,j)%an + aewnsp(i,j)%as +
aewnsp(i,j)%ap
    ……
  END DO
END DO
```

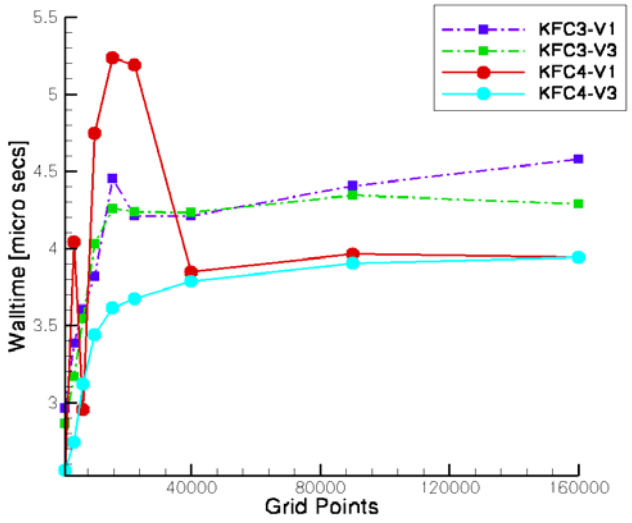
Figure 4-25 Using data structures in place of arrays in V3

Table 4-3 Normalized walltime values (in micro seconds) for V1 and V3 on KFC3

Grid size	Walltime - V1	Walltime - V3
30 x 30	2.96	2.86
60 x 60	3.38	3.17
80 x 80	3.60	3.54
100 x 100	3.82	4.03
125 x 125	4.45	4.26
150 x 150	4.21	4.24
200 x 200	4.21	4.23
300 x 300	4.40	4.34
400 x 400	4.58	4.29
500 x 500	4.49	4.30
600 x 600	5.30	4.22
800 x 800	6.11	------
900 x 900	6.24	------
1000 x 1000	6.45	4.41



(a)



(b)

Figure 4-26 Walltime as a function of grid size on KFC3 and KFC4 for V1 and V3 (a) for all grid sizes (b) zoomed plot

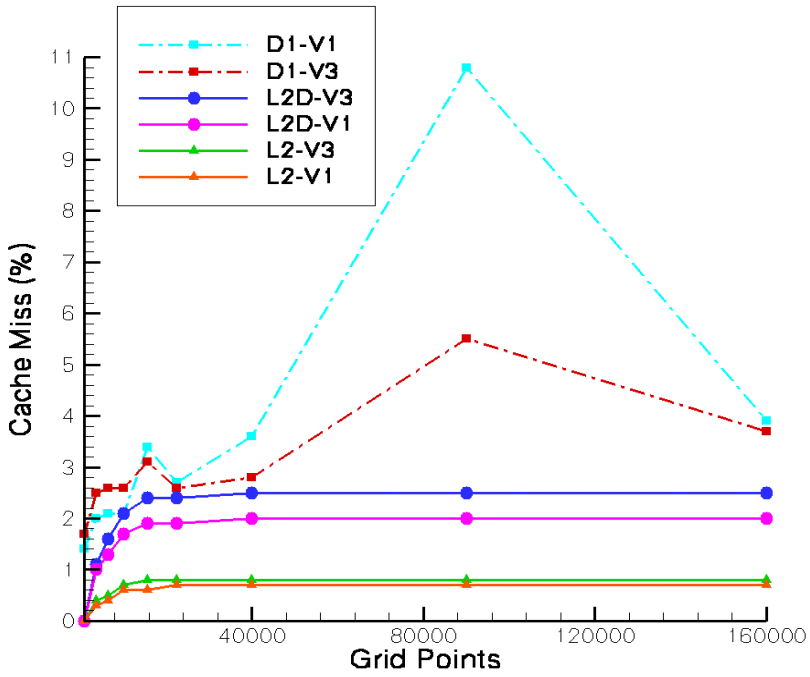
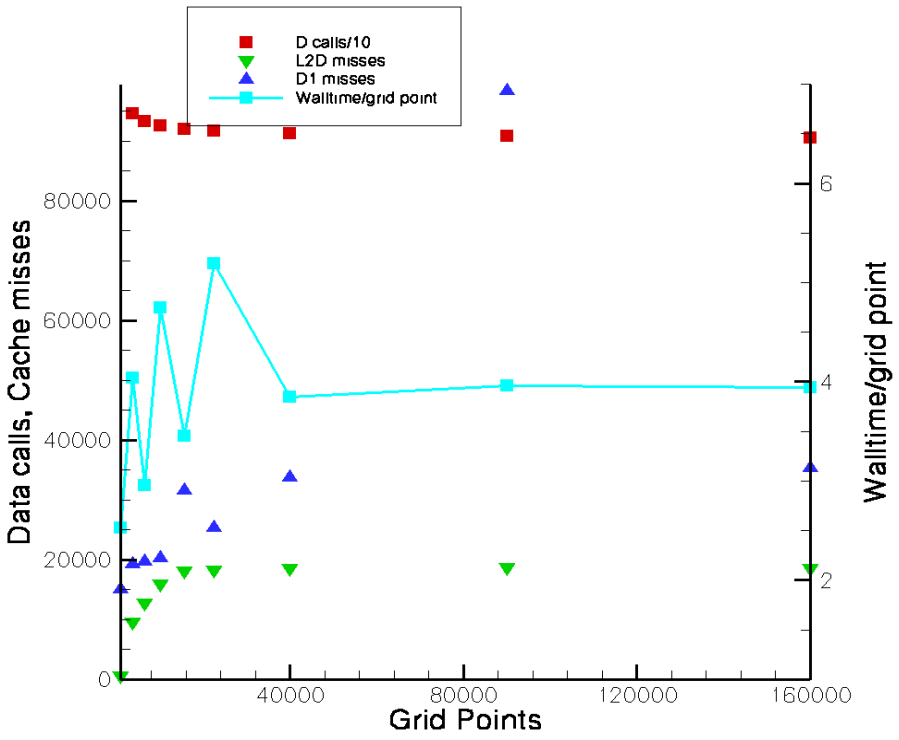
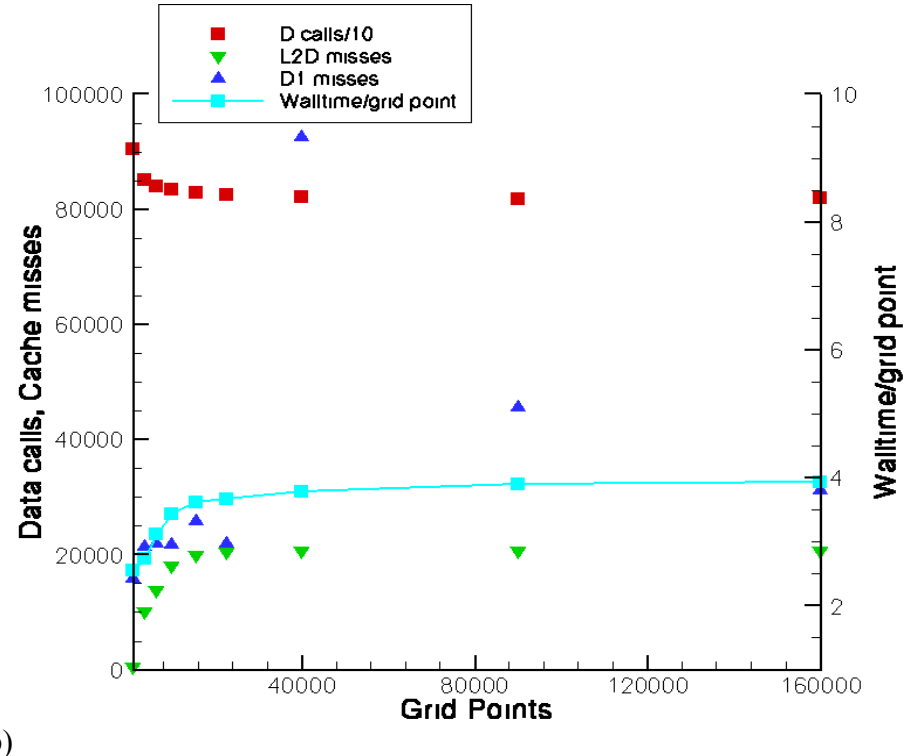


Figure 4-27 Comparison of D1, L2 and L2D cache miss rates for V1 and V3 on KFC4



(a)

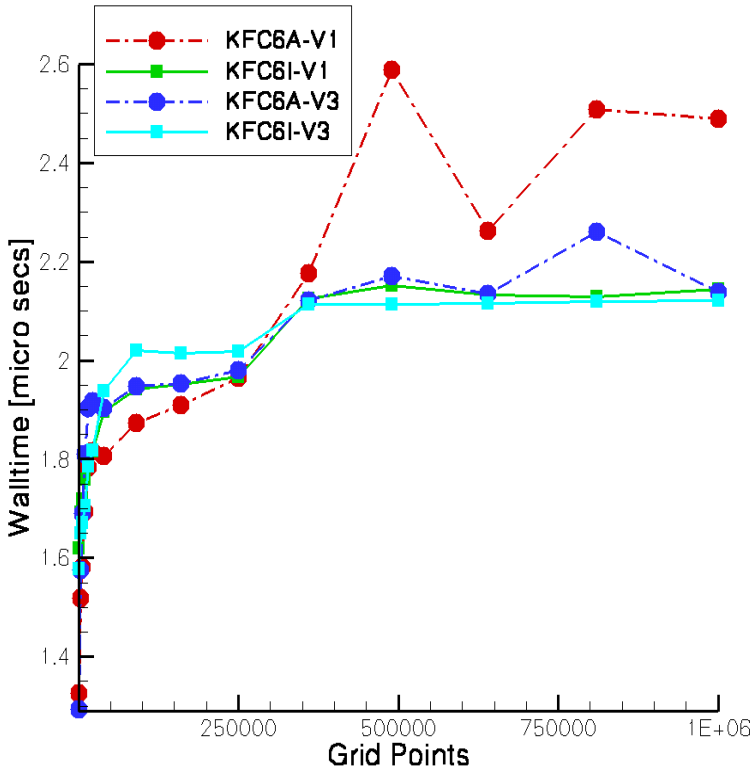


(b)

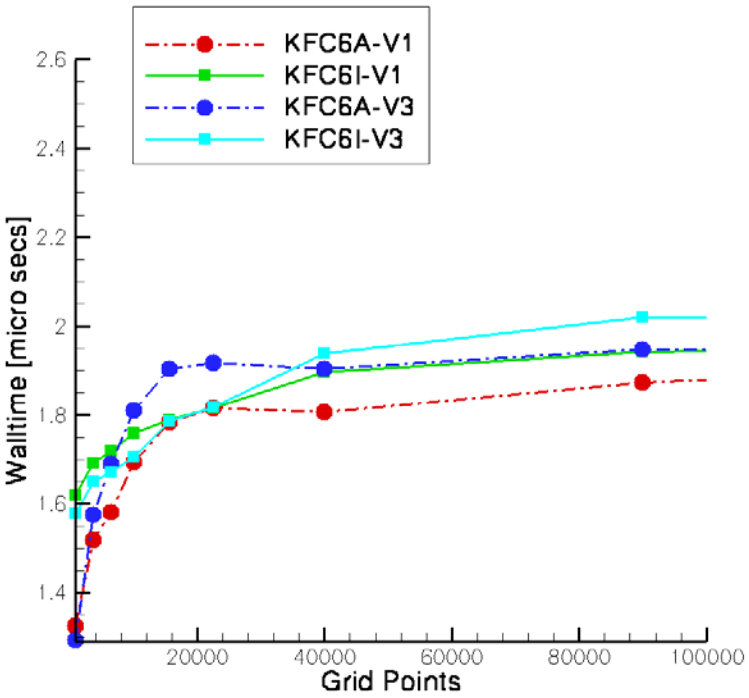
Figure 4-28 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC4 for GHOST (a) V1 (b) V3.

4.5.3.2 KFC6 Results

Results of walltime plot on KFC6A and KFC6I is presented in Figures 4-29 and b. Improvements on KFC6A vary from 5 to 10% for smaller grids (up to 400 x 400) while for larger grids, performance gains up to 15% are observed. As can be noticed, improvements on KFC6I are meager and can be attributed to faster processors and advanced hardware. In KFC6A, performance improvements in V3 are sporadic. As seen from Figures 4-29a and b, V3 on KFC6A performs better (gains of up to 16%) for larger grids starting around 250000 grid points probably because of benefits of using data structures are realized due to data size becoming exponentially large at such grid sizes. Valgrind results are presented in Figure 4-30. On KFC6I, cache miss rates are more in V3 than in V1 because of fewer (by 5%) data misses (Figures 4-31a, b) and this might be the reason for improvements in walltime. But, walltime behavior cannot be simply explained with cache miss data.



(a)



(b)

Figure 4-29 Walltime as a function of grid size on KFC6I and KFC6A for V2 and V3 (a) for all grid sizes (b) zoomed plot till 100000 grid points

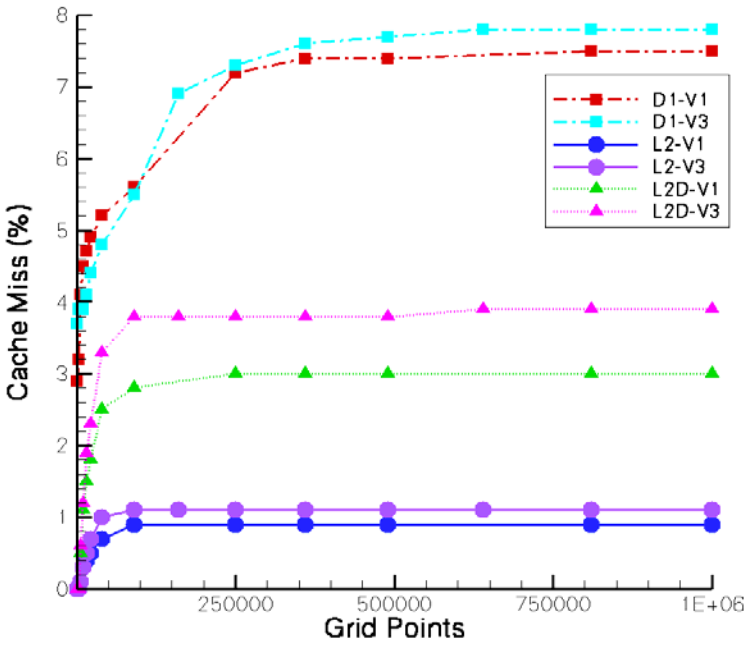
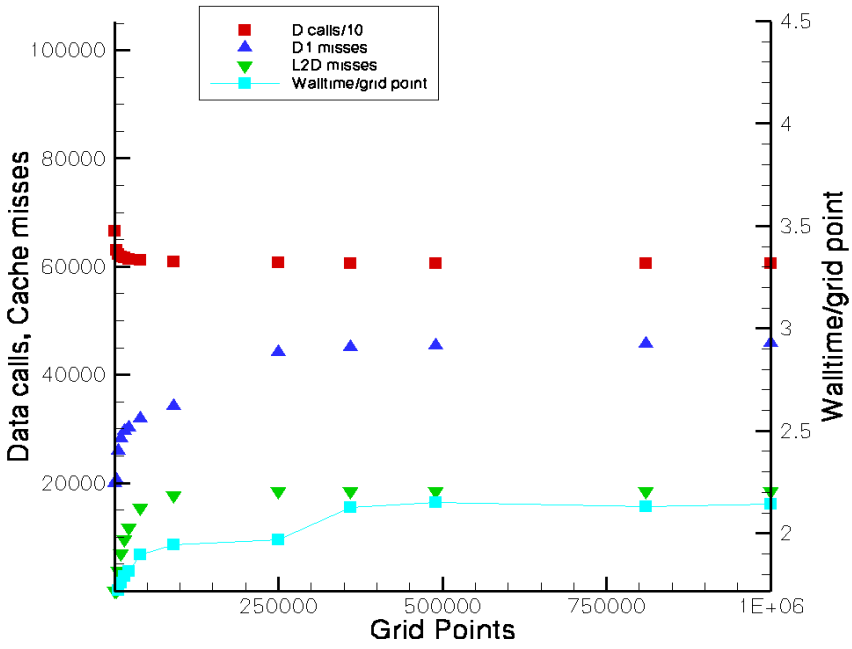
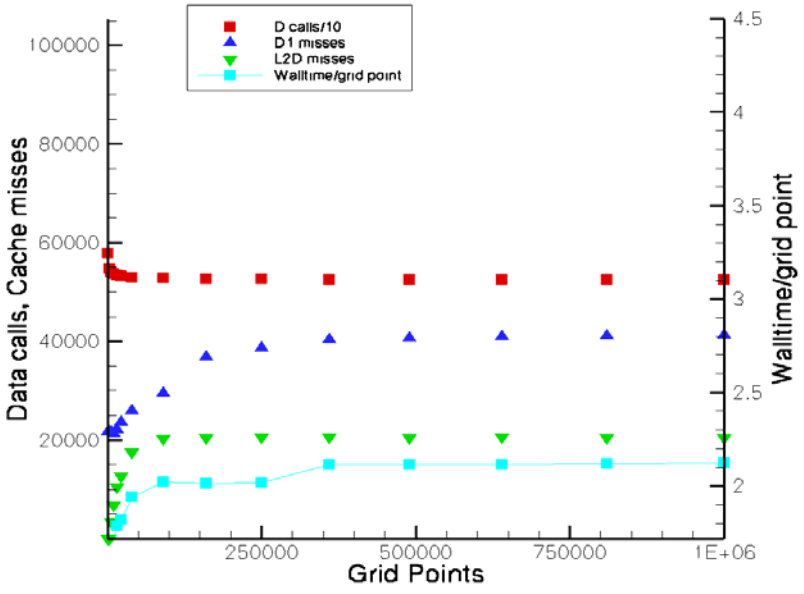


Figure 4-30 Comparison of D1, L2 and L2D cache miss rates for V1 and V3 on KFC6I



(a)



(b)

Figure 4-31 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC6I for GHOST (a) V1 (b) V3

4.6 SUMMARY OF OPTIMIZATION EFFORT AND RESULTS OF FURTHER EFFORTS OF TUNING GHOST

This section provides an overall analysis of results of optimization effort carried out on GHOST. The work presented above was carried out till December 2004. This work was incorporated into the paper by "R. P. LeBeau, P. Kristipati, S. Gupta, H. Chen, P. G. Huang, "Joint Performance Evaluation and Optimization of Two CFD Codes on Commodity Clusters", AIAA – 2005 – 1380, January 2005" [52]. This paper included further optimization efforts on GHOST beyond the scope of this thesis, but based on the work presented so far. This analysis focused on KFC3 and KFC4. This section presents these results along with summarizing the results presented above on those clusters.

In this optimization effort, the optimization steps were similar to those presented previously in section 4.4 except for step 6:

1) Replacing the allocation/de-allocation scheme with permanent variables
2) Correcting the orientation of the *i*,*j* sweeps to the cache-conserving form (outer loop *i*, inner loop *j*) consistent with the storage in memory.
3) Aggressive cleaning of redundant computations, unnecessary divisions, and other excessive mathematical activity.
4) Removing unneeded if-then structures, particularly on sweeps that do not encompass the full *i*,*j* grid.
5) Restructuring the variables from the single array form, $\phi_1(i,j)$ and $\phi_2(i,j)$, to an array of structures $\Phi(i,j):\phi_1,\phi_2$.
6) Applying a sub-blocking scheme in which a grid is divided into subgrids that effectively fit into the L2 cache.

Based on the above steps, the results presented above account for the three versions of GHOST *viz.* V1, V2 and V3. In order to have a deeper understanding of effect of applying each of the above discussed techniques, LeBeau *et al.* [52] presents more optimized versions of GHOST as shown below:

V4 – V0 + step 5 5 – original + step 6

V6 – original + steps 2-4, 6 V7 – original + steps 2-6

The standard LINUX tool *gprof* was used to determine the relative cost of each of the six key GHOST subroutines, which collectively involve over 95% of the total

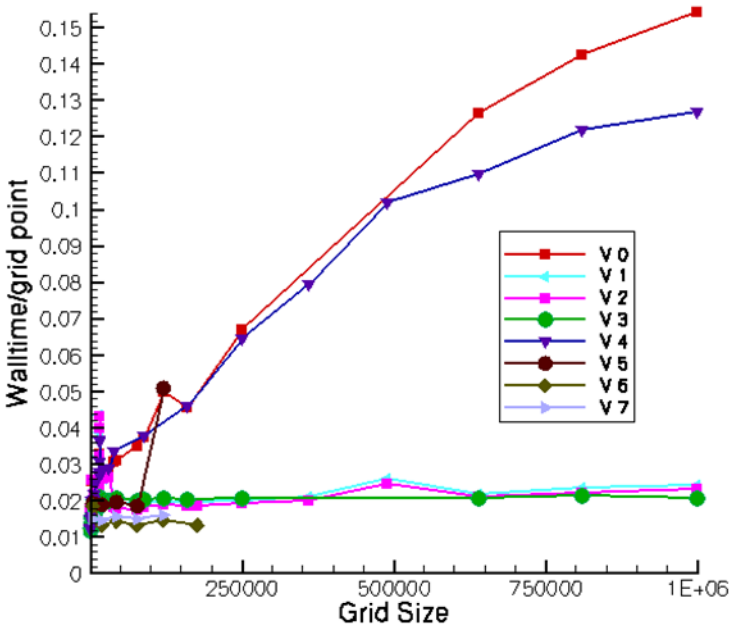
runtime. Table 4-4 presents the profile results over a 5000 iteration simulation for code versions 0-4 for four different grid sizes. As a result of coding improvements, the three most costly routines (*cont*, *tdma*, *quick*) shift positions relative to one another, while the other three key subroutines generally hold their relative positions and proportions. An exception to this is the application of the array of structures (step 5) which serves in the largest (900 x 900) grid to significantly reduce the *tdma* cost relative to the versions without step 5 (compare V0 and V4, V2 and V3). This is not unexpected, as the array of structures was designed based on the loops in *tdma*; however, for smaller grids the relationship to *tdma* is not as apparent, either leading to across the board improvements (90 x 90, V2 to V3) or a decline in *tdma* performance but improved performance in other subroutines (150 x 150, V2 to V3). Adding only step 5 to the original code was only clearly a benefit on the largest grid and in some cases proved detrimental, a fact reflected in the curves in Figure 4-32. This is not inconsistent with the results presented so far; it was observed that using the array of structures instead of arrays essentially did not improve performance but lead to a more consistent behavior of walltime plot as discussed in section 4.5.3.1.

Table 4-4 Walltime in seconds spent in key subroutines for GHOST on four grid sizes over 5000 iterations

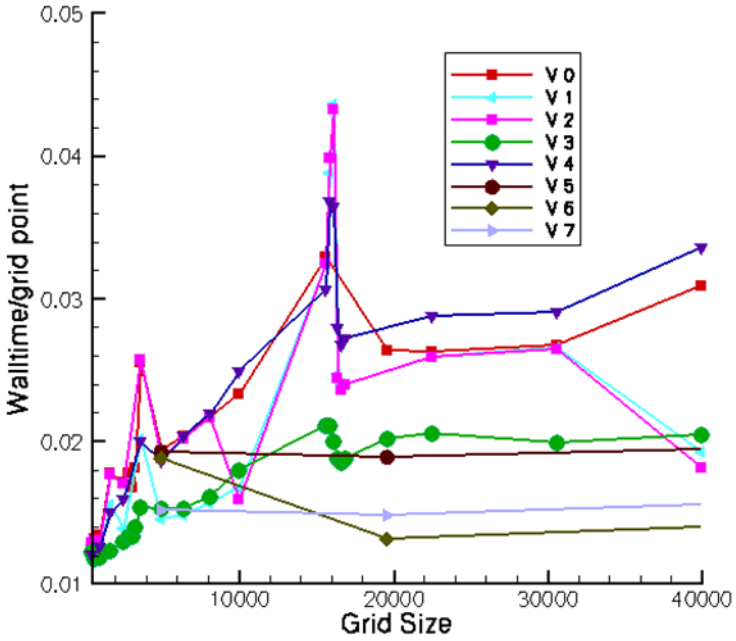
90x90	V0	V1	V2	V3	V4		150x150	V0	V1	V2	V3	V4
Total	174.0	127.1	177.4	131.7	181.9		Total	590.3	593.1	588.7	452.3	658.9
Cont	53.4	42.2	54.5	41.2	53.5		*cont*	170.8	170.5	170.4	129.4	166.8
Quick	37.5	23.6	38.5	29.4	40.1		*quick*	111.2	113.6	111.1	81.1	126.8
Tdma	25.3	24.6	25.9	19.3	20.0		*tdma*	99.9	100.3	99.9	114.7	119.4
cal_prop	21.4	13.9	22.0	14.4	21.9		*cal_prop*	82.2	83.7	82.1	46.5	82.7
cal_v	14.9	9.2	15.2	10.4	18.3		*cal_v*	54.0	52.9	52.5	30.7	64.5
cal_u	14.8	8.5	14.8	10.0	17.6		*cal_u*	52.6	54.4	54.0	31.56	66.0
other	6.8	5.1	6.7	7.0	10.5		*other*	19.6	17.7	18.8	18.3	32.9

300x300	V0	V1	V2	V3	V4		900x900	V0	V1	V2	V3	V4
Total	3278	1736	1656	1849	3344		Total	48584	-	17239	16750	37269
Cont	930	496	425	510	887		*Cont*	15339	-	4309	4808	11305
quick	664	258	261	319	702		*quick*	6901	-	2697	3011	6108
tdma	509	494	489	502	510		*tdma*	5371	-	5673	4127	4273
cal_prop	484	201	203	202	476		*cal_prop*	6582	-	1980	1992	6687
cal_v	322	113	110	121	330		*cal_v*	7261	-	1047	1083	4817
cal_u	308	122	117	123	334		*cal_u*	6688	-	1081	1098	3559
other	61	51	52	72	104		*other*	443	-	452	631	520

The direct comparison between L2D miss rate and normalized walltime is shown in Figures 4-33a and b. The L2D miss rate largely parallels and therefore likely explains the overall shape of the curve for the larger grids, but the smaller grids (less than 200 x 200) and for the scaling and the secondary variations in the larger grids, an explanation based solely on L2 cache is inadequate.

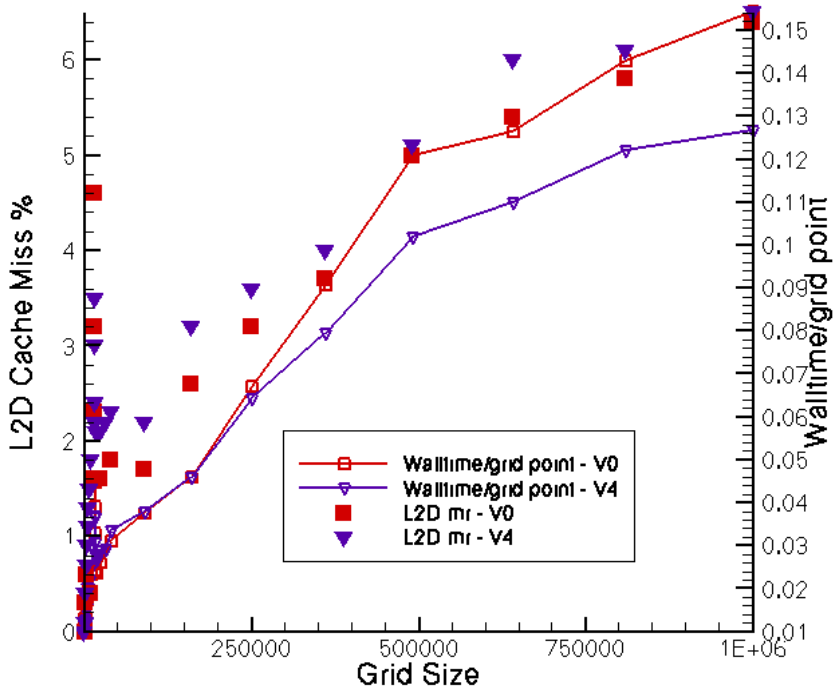


(a)

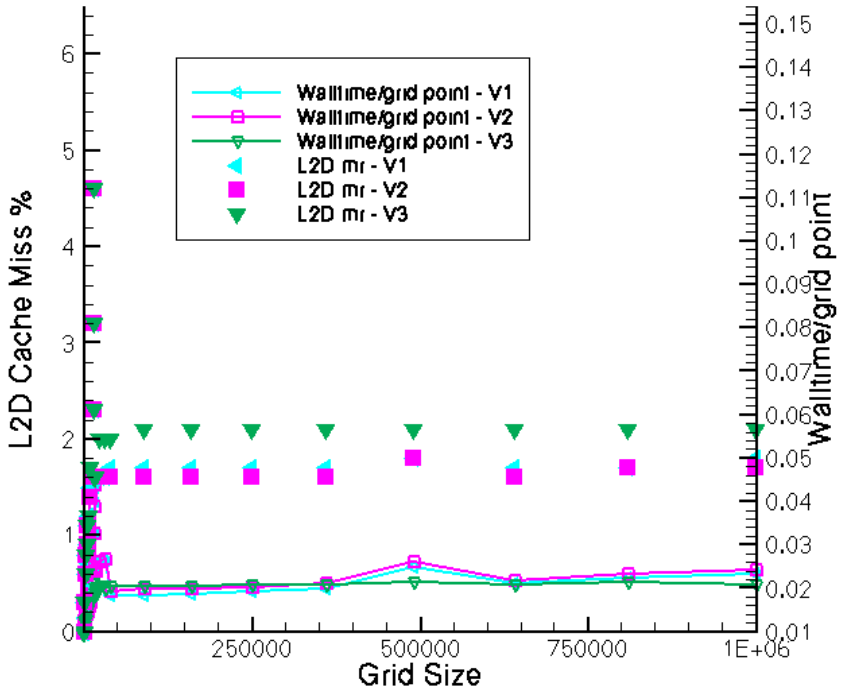


(b)

Figure 4-32 Overall performance of the eight versions of GHOST in terms of walltime/gridpoint versus grid size with (a) the full range and (b) the more complicated region for grids smaller than 200 x 200



(a)



(b)

Figure 4-33 Comparisons of L2 cache miss rate and normalized walltime versus grid size on KFC4 for GHOST (a) Version 0 and Version 4, (b) Versions 1-3

Instead, as discussed in the above section, the overall performance is also tied to the L1 data cache miss rate (D1), which varied in an unpredictable fashion on the smaller grids, and the number of total data calls required, which could vary significantly from code version to code version for the same grid size particularly with the addition of the variable array structure (step 5) or sub-blocking (step 6). Several walltime features that are not clearly explained by L2D misses can at least be qualitatively explained by the other two features. Sharp peaks in the D1 cache misses create much of the erratic behavior on the small grids as in the case in Figure 4-28, as well as generating the small oscillations in normalized walltime for large grids in version 2 for example near 500,000 grid points in Figure 4-22. So, the combination of all three types of cache activity does provide an explanation of most of the walltime profile features, confirming the strong effect of cache efficiency on code performance.

Several conclusions are drawn from this analysis. The first is that the increase of cache miss rate with grid size was largely a function of poor loop construction and was solved for large grids (greater than 200 x 200) by step 2 and that this improvement by

itself yields dramatic gains in performance at these large grid sizes. The addition of steps 3 and 4 further decreases normalized walltime, if not as dramatically, for large grids. Step 5 also provides an incremental performance boost for the largest grids (greater than 600 x 600) but for intermediate grids it is slightly detrimental. For grids smaller than 200 x 200, the performance is strongly influenced by variations in the L1 cache miss rate as well as L2. The result is a strongly variable normalized walltime and cache miss rate, with strong peaks in both between grid sizes of 125 x 125 and 130 x 130, as well as smaller peaks elsewhere. This behavior largely eliminates the benefits of optimization steps 2-4 at many grid sizes, but the addition of Step 5 appears to reduce much of the erratic behavior in this region, yielding a much smoother set of normalized cache and walltime curves.

4.6.1 OVERALL PERFORMANCE

Revisiting the speedup plots in Figure 4-34, it is clear that the optimized versions of the code do not exhibit significant superlinear speedup but either near-linear or for smaller grids a sublinear curve, confirming that the dramatic superlinear speedup was a result of cache effects. Alternatively, one can examine the walltime data starting with the fact that for the smallest grids (30 x 30 and smaller) for all non-sub-blocked versions the L2D cache miss rate is less than 0.1% and the D1 miss rate is typically less than 2%. This suggests that normalized walltimes for these small grids are close to the best possible cache-driven performance (where in theory the total cache miss rate would be 0%). The absolute best normalized walltime of the cases considered is for version 3 on a 30 x 30 grid at 11.8 ms/gridpoint for 5000 iterations. If all the normalized walltime values are divided by this optimal value as in Figure 4-35, the result shows how close to ideal each code version is. The best optimized code (version 3) is within a factor of 2 of this best result across all grid sizes. This is considerably better than the original code, which can be as much as 13 times slower that the theoretical optimum. The limited sub-blocking data stays within about 30% of the best value, with the exception of the largest grid tested with the original code plus sub-blocking (version 5).

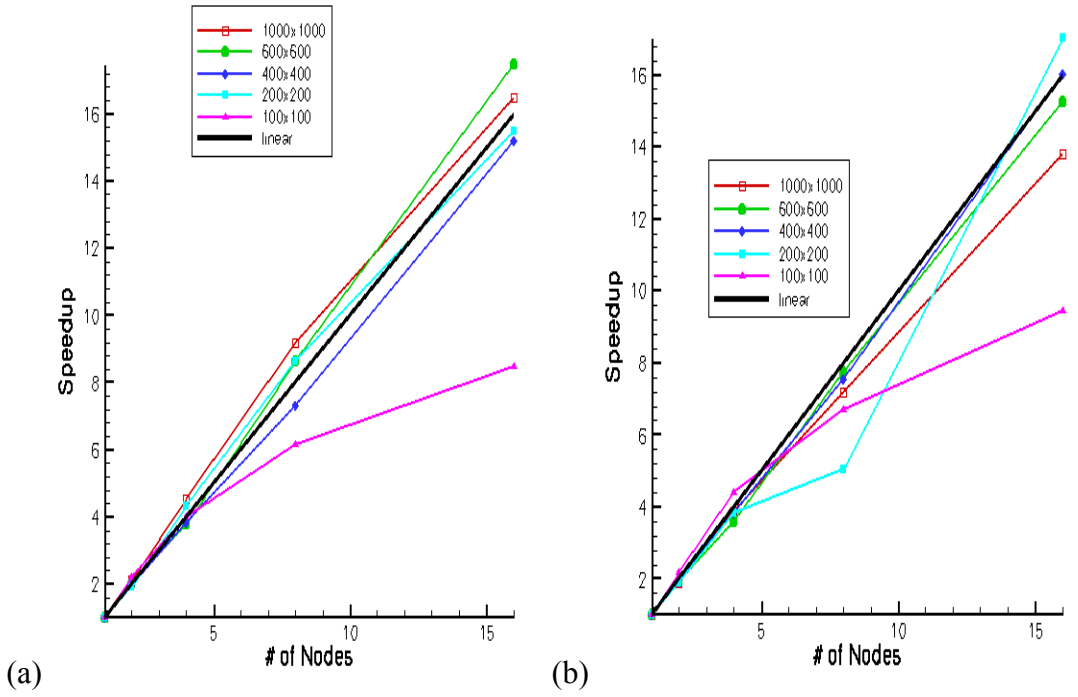


Figure 4-34 Speedup of GHOST on KFC4 for grids of varying size with (a) Version 2 and (b) Version 3

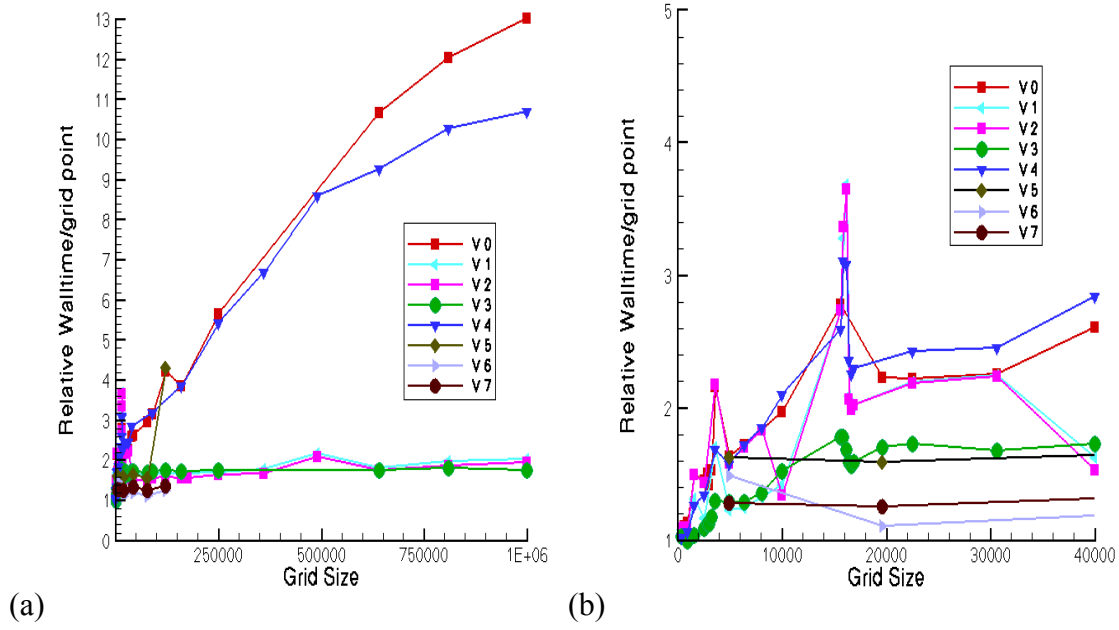


Figure 4-35 Overall performance of the eight versions of GHOST relative to the "optimal" value of 11.8 ms/gridpoint for 5000 iterations with (a) the full range and (b) grids smaller than 200 x 200

4.7 ACCURACY RESULTS

Accuracy tests conducted by A. Palki [53] showed that the results were unchanged by the code changes and the final results are in agreement with Ghia *et. al.* [66].

4.8 SUMMARY AND FURTHER WORK

This chapter presented the results of applying various techniques to optimize the performance of GHOST on commodity cluster architectures. It was observed that the best optimized version (V3) of the code was within a factor of 2 of the estimated optimal performance over all the tested grid sizes and the overall performance improvement for this case relative to the original code ranged from 20% faster for small grids to over 6 times faster for the largest. In next chapter, results of External and *Internal* blocking, (a new technique that will be introduced in next chapter) are presented along with results of further optimization effort.

CHAPTER - 5

5. STAGE TWO PERFORMANCE TUNING RESULTS

As discussed in chapter 4, the best tuned version of GHOST was with in a factor of 2 of the estimated optimal performance over all the tested grid sizes and the overall performance for this version ranged from 20% faster for smaller grids to over 6 times faster for the largest. After the tuning effort that was presented in chapter 4 was carried out, Palki [53] did extensive investigation and optimized the performance of GHOST by applying external and internal blocking techniques on V3. This chapter summarizes the results of the optimization effort carried out by Palki [53] to show the performance of V0 and V3 when these techniques are applied along with problems associated with these techniques. Later, results of stage two of optimization effort carried out in this work are presented.

5.1 EXTERNAL BLOCKING

As discussed earlier, external blocking involves the breaking the computational grid into smaller sized cache friendly blocks. This step is carried out during the grid generation process and is done by the grid generator code g.f90 introduced in chapter 3. This section reviews external blocking results compiled by Palki [53] on the best optimized version (V3) of GHOST for cavity flow problem that was described in chapter 4.

5.1.1 KFC4 AND KFC5 RESULTS

The walltime (normalized by grid size and iterations) plot for the tuned V3 code on KFC4 is shown in Figure 5-1(a). For comparison, a similar plot is shown for the V0 code in Figure 5-1(b). In the case of V3 (because of the optimization effort), the walltime for the unblocked (single zone) code does not increase with an increase in the grid size. The performance of V0 when external blocking is applied is similar to the performance of V3 without external blocking. The optimization effort thus relieves the burden on the programmer to split the grid into cache sized blocks so as to realize improved performance. Still, external blocking applied to V3 yields more favorable results. As presented in Table 5-1, in the case of V3, external blocking on a 600 x 600 grid yields an

improvement of approximately 30% in walltime and this value remains the same throughout the block sizes (30 x 30, 40 x 40 and 50 x 50) tested. This is a noticeable difference between V0 Ext and V3 Ext (V0 and V3 with external blocking applied respectively) because walltime for V0 Ext was largely dependent on the subblock size as discussed in chapter 5. This further relieves the burden from the programmer in that while splitting the grid a relatively larger subblock size can be chosen leading to fewer zones, meaning less complexity in designing the input file and fewer ghost points leading to fewer data calls. In the case of V3, the performance of the externally blocked grid is also highly scalable and unchanging over a wide range of grid sizes i.e., the normalized walltime tends to stay almost constant irrespective of the grid size. Hence the actual normalized speed of a blocked computation is comparable to the single grid computation of the same size for reasonably large (at least till tested 70 x 70 grid) computational problems. Similar walltime results on KFC5 are presented in Figure 5-2. Table 5-2 presents improvements in walltime for a 600 x 600 grid on KFC5 because of external blocking applied to V3.

Table 5-1 External Blocking results for 600 x 600 grid on KFC4 with various subgrids[53]

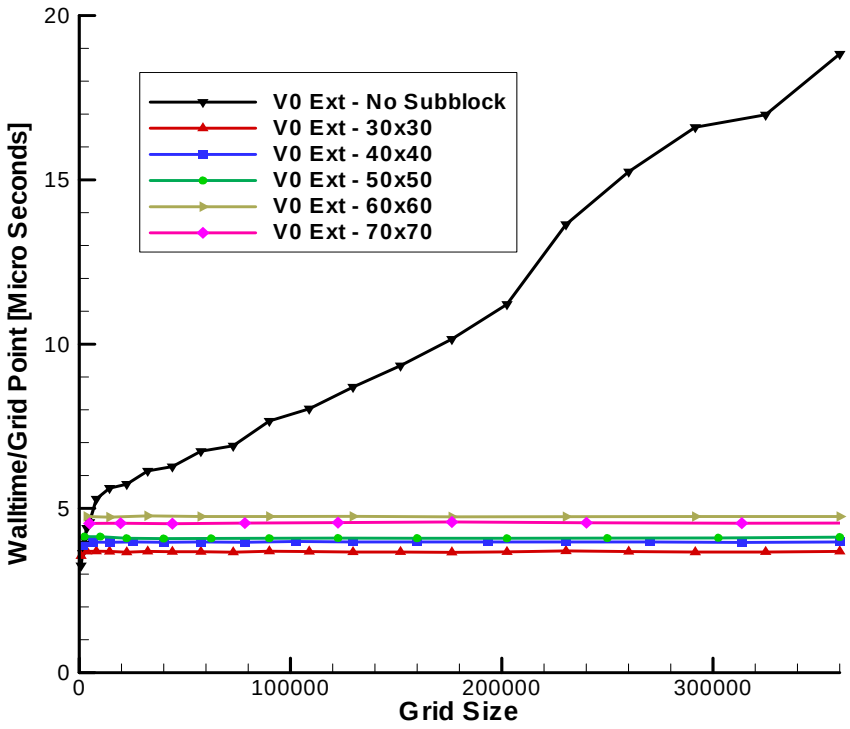
Block Size	Walltime/Grid Point/Iteration [μ secs]		% Improvement compared to No Block	
	V0	V3	V0	V3
No Block	18.83	4.63	-	-
70 x 70	4.55	3.48	75.8%	24.8%
60 x 60	4.75	3.40	74.7%	26.56%
50 x 50	4.12	3.33	78.1%	28.07%
40 x 40	3.97	3.31	78.9%	28.5%
30 x 30	3.69	3.29	80.4%	28.9%

Table 5-2 External Blocking results for 600 x 600 grid on KFC5 with various subgrids[53]

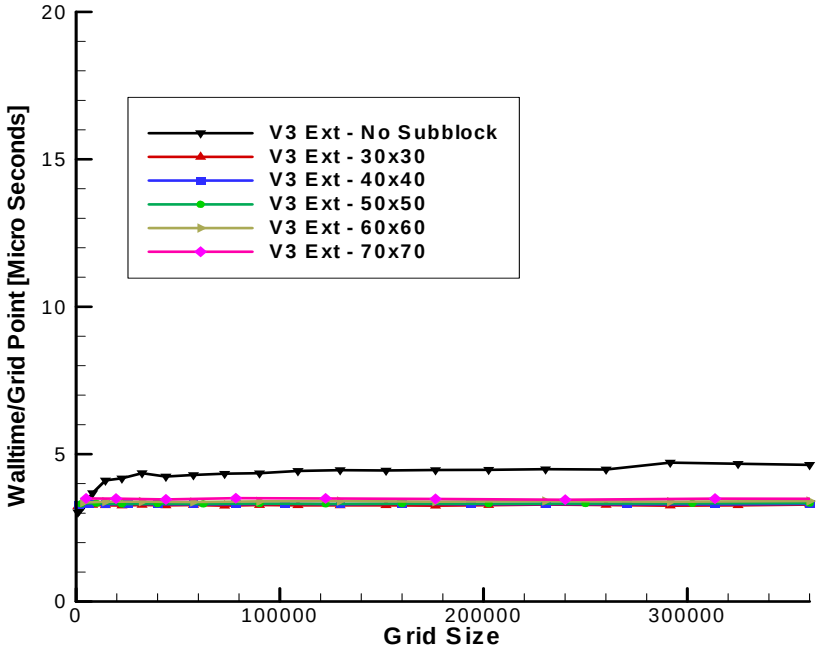
Block Size	Walltime/Grid Point/Iteration [μ secs]		% Improvement compared to No Block	
	V0	V3	V0	V3
No Block	6.51	2.41	-	-
70 x 70	2.32	1.89	64.3%	21.5%
60 x 60	2.31	1.86	64.5%	22.8%
50 x 50	2.18	1.78	66.5%	26.1%

40 x 40	2.03	1.76	68.8%	26.9%
30 x 30	1.98	1.74	69.5%	27.8%

As expected, due to better and faster processors the overall code performance is improved for the unblocked grid on KFC5 compared to KFC4. The potential gain obtained from improving cache performance is reduced due to the greater bandwidth and better memory hierarchy structure. With the application of 30 x 30 subblocks for a 600 x 600 grid on KFC5, a decrease of 69% was observed for the V0 code and 27.8% for the V3 code when compared to the unsubblocked code as potential gains had already been realized in V3 because of applying code optimization techniques. But, it is important to note that subblocking applied on V3 leads to better results than on V0 as V3 is already improvised version of V0 (as discussed in chapter 4). Thus, in order to realize the full benefits of external blocking technique, the code must be optimized on a single node first, the details of which are presented in chapter 4.

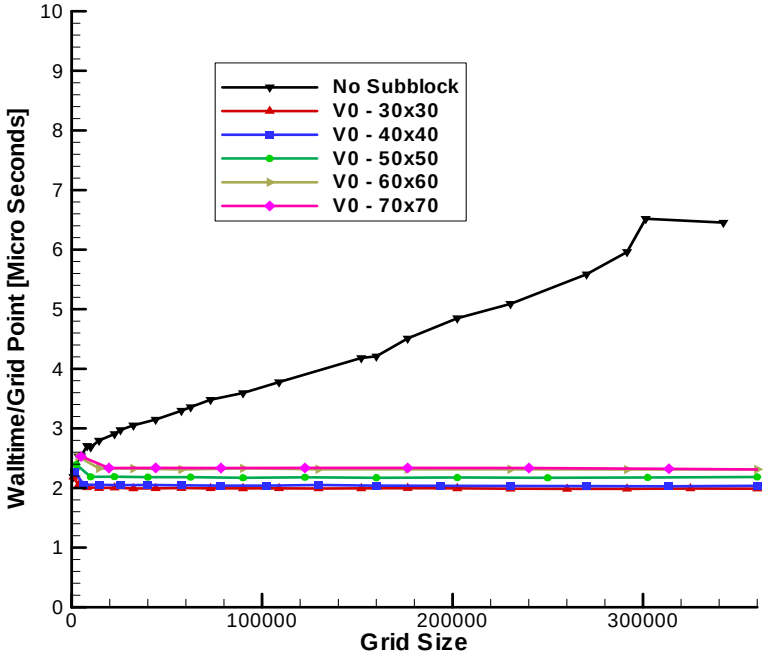


a)

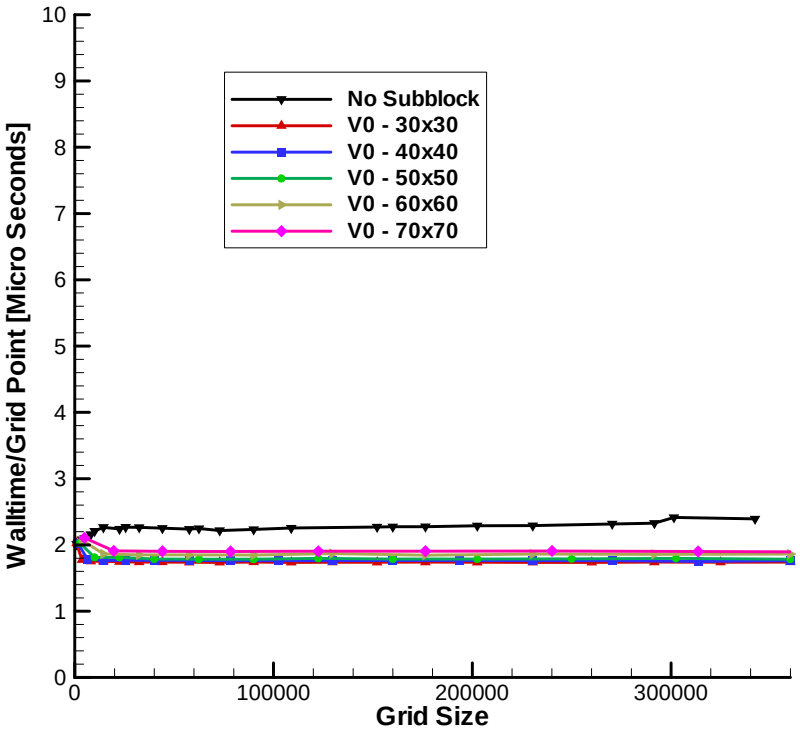


b)

Figure 5-1 External Blocking - Walltime as a function of grid size on KFC4 for the lid-driven test case. (a)V0 (b)V3 [53]



a)



b)

Figure 5-2 External Blocking - Walltime as a function of subgrid size on KFC5 for the lid-driven test case. (a) V0 (b) V3 [53]

Although external blocking can yield impressive performance improvements (as shown above), a major disadvantage of this technique is the difficulty in implementing it, since it is not easy to break up a grid with a complicated geometry or boundary conditions into smaller blocks. Creating a viable automated system to split the grid led to the idea of internal blocking technique, which is an automated version of external blocking.

5.2 INTERNAL BLOCKING

The underlying principle behind internal blocking is quite similar to that of external blocking. It involves breaking up the grid into smaller cache fitting blocks, solving the governing equations on these smaller blocks, and then putting them back together before the start of the MPI communications to get the overall solution. As discussed at the beginning of this chapter, Palki [53] did extensive investigation on this

technique and developed an automated process to split the grid into smaller blocks that fit into L2 cache of the processor on which calculations are being carried out.

This technique has been presented in Figure 5-3. The grid presented is that of an airfoil at a certain angle of attack. In order to decrease the computation time through parallel processing, the grid had already been split into multiple blocks (external blocking). The flow field across each of the blocks can be solved on a different processor. As discussed above, internal blocking involves the splitting of each of these individual blocks into cache sized blocks. This has been illustrated with the help of a magnified view of the subblocks in Figure 6-3.

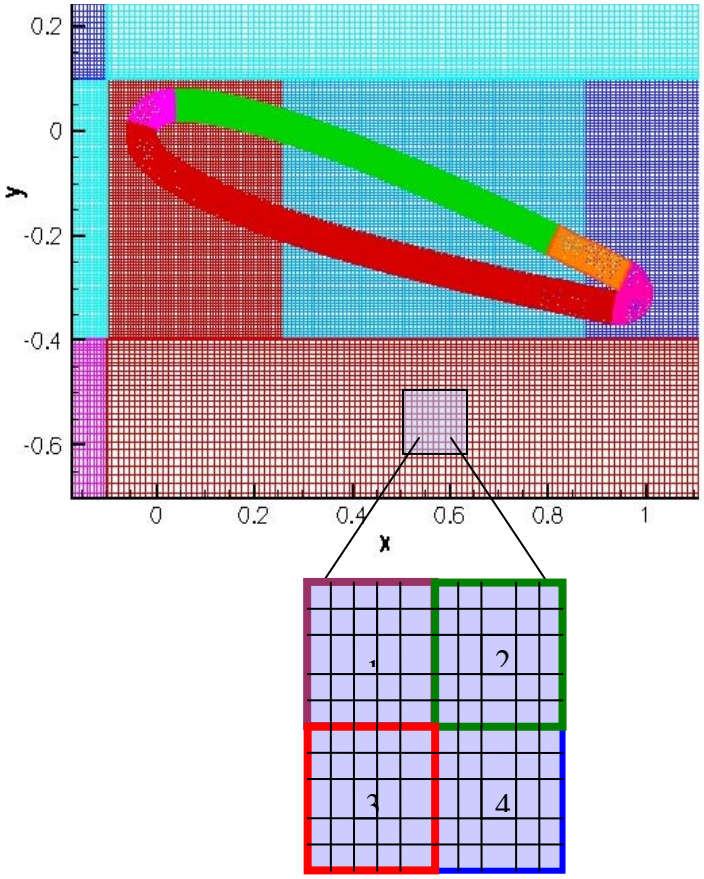


Figure 5-3 Illustration of Internal Blocking [53]

5.2.1 IMPLEMENTATION OF INTERNAL BLOCKING IN GHOST

In order to implement internal blocking into GHOST, four additional subroutines (*Internal_block, break_velocity, scalc_flowfield, combine_velocity*) have been added [53]. All the information pertaining to a grid such as the *x, y* coordinates of the grid points are stored in arrays. The subroutines split these arrays into smaller sized arrays such that each array will hold all the necessary information pertaining to a smaller sized block. This is equivalent to breaking up the grid into smaller sized, cache fitting blocks. Since the grid parameters remain constant for a given grid, this operation needs to be performed only once. The flow variables such as *u*, *v*, and *p* are also stored in arrays. Unlike the grid parameters, the values of these variables are updated every iteration. These updated values are required to calculate the values across the artificial boundaries. Hence, the subroutines split up the arrays consisting of these flow variables at the start of the calculations and then put them back together before the beginning of the MPI communication.

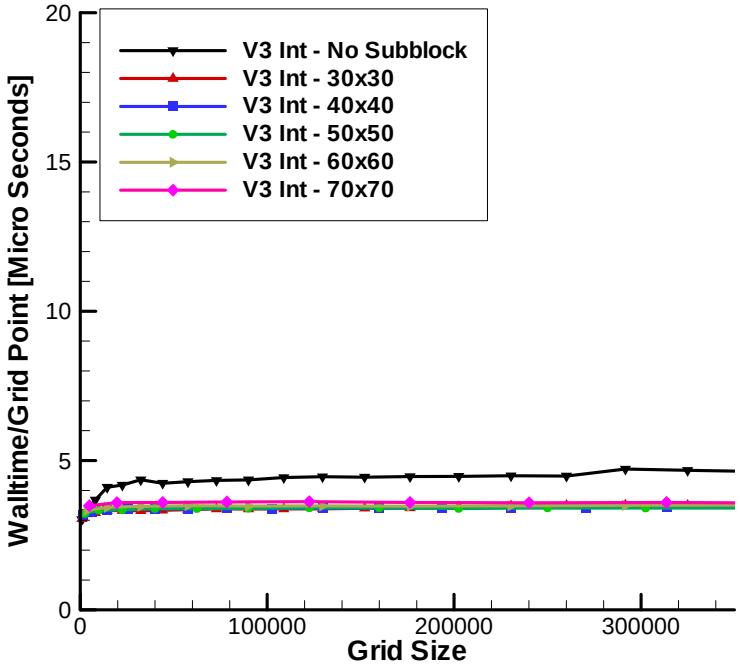
5.2.2 KFC3 AND KFC4 RESULTS

A plot of walltime as a function of subgrid size for the untuned V0 code on KFC4 is shown in Figure 5-4(a). As mentioned in chapter 4, walltime is the average time for a single iteration over a 5000 iteration simulation and it is normalized by the number of grid points to eliminate the effect of increasing walltime with increasing grid sizes.

Walltime behavior with internal blocking applied closely matches with the one with external blocking. Walltime for “V0 Int - No Subblock” (V0 with no subblocking) increases with increasing grid size. Even with internal subblocking, walltime for V0 increases with subblock size and largely depends on subblock size. The best performance improvement was obtained with the 30 x 30 subblock grid. In the case of 600 x 600 grid, walltime with 30 x 30 subblock is lower by a factor of 4.75 compared to the unsubblocked grid. The walltime plot for the tuned V3 code on KFC4 is shown in Figure 4-3(b). There is a decrease of 23.5% when compared to the unsubblocked code. Walltime values for 600 x 600 grid for various subblock sizes (with Internal Blocking) are shown in Table 5-3.

Table 5-3 Internal Blocking results for 600 x 600 grid on KFC4 with subgrids of various sizes [53]

Block Size	Walltime/Grid Point/Iteration [μ secs]		% Improvement compared to No Block	
	V0	V3	V0	V3
No Block	18.83	4.63	-	-
70 x 70	4.74	3.57	74.82	22.89
60 x 60	4.43	3.50	76.47	24.406
50 x 50	4.24	3.40	77.48	26.56
40 x 40	3.98	3.44	78.86	25.7
30 x 30	3.93	3.53	79.12	23.75



b)

Figure 5-4 Internal Blocking Results - Walltime as a function of subgrid size for GHOST on KFC4 for the lid-driven test case (a) V0 (b) V3 [53]

With internal blocking, for V0 the best performance is obtained with 30 x 30 blocks, while for V3, 50 x 50 blocks showed the best performance, although for the V3 code the difference in the value of the walltime is quite small for the various block sizes. Hence the performance improvement obtained by splitting up the grid using only 30 x 30

blocks will be almost the same as a grid broken up using blocks of sizes varying from 30 x 30 to 70 x 70. Also, walltime values scale well through out the problem sizes tested for all subblocks presented in the Figure 5-4.

5.2.3 ACCURACY TEST RESULTS

Although the internal blocking technique did not pose any problems in terms of accuracy of results for cavity flow test case (presented in chapter 4), the results were not accurate [53] for a more complicated test case when the grid was subblocked in the same way as in cavity flow problem. Accuracy tests were conducted for flow over a NACA 4415 airfoil test case with Reynolds number of 100,000 and a time step dt = 0.0025. Inaccurate results were attributed to probable coding error in the way the boundary conditions were implemented. In addition to the problems in terms of the accuracy of the results, the internal blocking approach requires a fair bit of change in the underlying algorithm of the code. Also, from results in chapter 4, it was noticed that L2D cache miss rates of 2% are still persistent in the most optimized version (V3) of GHOST. This presented an opportunity for more tuning effort on V3 with the aim of reducing the cache miss rates to the maximum extent possible, without relying on subblocking. Results of further tuning effort on GHOST are presented in the next sections.

5.3 FURTHER TUNING EFFORT ON GHOST

A detailed review of V3 revealed the possibility of implementing the following steps:

1) Subroutine inlining

2) Additional cleaning of redundant computations, unnecessary divisions and other excessive mathematical activity.

In step 1, the inlining technique, discussed in chapter 2, was applied on subroutine *quick_struct*. The algorithm of this subroutine was inlined into the code by replacing it with the function call. The reason for choosing this subroutine was because this subroutine is called only twice in a given iteration and because of this, it is fairly easy to implement the inlining technique. Step 1 proved ineffective resulting in neither a significant change in walltime or cache performance and as such is left out of the subsequent analysis.

Step 2 has similar techniques that were implemented in V2. As discussed in chapter 4, the tuning effort for V2 was focused on subroutine *cont* as such other subroutines were unchanged. It was observed that advanced architectures (like KFC6) benefited (decrease in data calls, data misses and L2D misses) more from changes in V2 than changes in V3 while relatively older architectures (like KFC3 and KFC4) benefited significantly from almost all of the optimization techniques discussed in chapter 4. This observation directed further optimization effort to focus on reducing redundant mathematical activity so as to reduce number of data calls. The cavity flow test case is the test case on which further optimization effort was carried out and Valgrind was used to analyze cache behavior of the code on different machines.

5.4 RESULTS OF FURTHER TUNING EFFORT

As discussed in chapter 4, initial performance tuning efforts primarily resulted in three major versions of the original GHOST code. They are V1, V2 and V3. Further optimization efforts [52] resulted in V4, V5, V6 and V7. But, as presented in chapter 4, the most optimized version of GHOST was considered V3 and so further performance optimization effort on GHOST was base lined from V3 and as such V8 is an improvised version of V3.

5.4.1 VERSION 8 OR V8

Version 8 or V8 is the result of stage two of performance tuning effort. As discussed above, the focus in this stage was to carry out additional cleaning of redundant computations, unnecessary divisions and other excessive mathematical activity by incorporating changes without modifying the underlying algorithm. These are discussed in the next few paragraphs.

5.4.1.1 Changing the Order of Condition Check in IF statements

As discussed in chapter 2, *'IF'* statements slow down a program for several reasons. Some of them are:

- Compiler can do fewer optimizations in their presence, such as loop unrolling
- Evaluation of the conditional takes time

- The continuous flow of data through the pipeline is interrupted when branching.

Often, the performance impact of '*if*' statements can be reduced by restructuring the program. It was observed that in V3, subroutines *cal_u, cal_v* and *cont* have a conditional check that might be improved with a minor change in the order of the condition check. It is well known that in most of the programming languages, the comparisons *equal* and *not equal* are faster than the comparisons *less than* and *greater than*. Although resultant improvements might be minor, this technique was implemented in three subroutines *cal_u, cal_v and cont*. This is presented in Figure 5-5.

Subroutine *cal_u, cal_v* and *cont* in V3:

```
IF (vol(i, j) < 1.e-20 .OR. inx(i, j) == 1) THEN
…….........
END IF
```

Subroutine *cal_u* in V8:

```
      ! order of condition check reversed
IF (inx(i, j) == 1 .OR. vol(i, j) < 1.e-20) THEN
…….........
END IF
```

Figure 5-5 Correcting the order of a conditional statement

5.4.1.2 Implementing Reciprocal Cache

As discussed in chapters 2 and 4 division operations have high latency. In V3, a few more areas (subroutines *cal_u, cal_v*) were identified where repeated division operations can be replaced by pre-calculated reciprocal values, a technique that was implemented in subroutine *cont* in V2. There were 10 instances in total where repeated division operations were replaced by a pre-calculated value that was calculated outside nested loop. Repeated divisions in V3 were being done inside nested *i, j* loops that span across the dimensions of the grid. As subroutines *cal_u* and *cal_v* are executed in each iteration, repeated divisions were being done for every iteration. For example in a 600 x 600 grid, this means in a given iteration, 10*600*600 divisions were being done in subroutines *cal_u and cal_v.* With the high cost of division operations (details discussed in chapter 2), this was a bottleneck for optimum performance of these subroutine in V3.

In V8, the reciprocal values calculated only once at the beginning of subroutines *cal_u* and *cal_v* are used throughout their code avoiding further division operations. This essentially mean 10 repeated division operations per iteration have been replaced by two (one per subroutine) division operations per iteration. This reduction in the number of division operations leads to savings in machine cycles. For example, cycle time for a division operation on AMD 64-bit architectures is 71 cycles while on Intel 64-bit it is 161 cycles [30] and because of this variation in cost of division operations between architectures, actual gains depend on architecture of a machine on which *Reciprocal Cache* is being implemented.

5.4.1.3 Loop Merging

As discussed in chapters 2 and 4, loop merging encourages *temporal locality* as the number of iterations that separate successive accesses to a given reused data is reduced if same data elements were being referenced in two consecutive loops before their merge. Figure 5-6 presents implementing reciprocal cache and loop merging techniques in subroutine *cal_u*. The fact that two nested loops were separated by a call to subroutine *tdma_struct* did not prevent us from implementing loop merging because the data elements inside the nested loops were not being used in the subroutine *tdma_struct* and so loop merging did not affect the integrity of the code and thus accuracy of the results were unaltered.

5.4.1.4 KFC3 and KFC4 Results

Walltime plots for V8 on KFC3 and KFC4 are presented in Figures 5-7a and b. Walltime values are compared with the ones for V3 as discussed earlier, V8 is improvised version of V3. Performance gains in V8 are more pronounced in KFC3 than in KFC4 although changes in V8 introduce spike in walltime value at 500 x 500 grid on KFC3. The walltime value for this particular grid is 30% more in V8 when compared to V0. Otherwise, performance gains up to 25% are realized for the remaining grids. The cause of the spike in KFC3 is unknown as Valgrind data was not collected on KFC3. On KFC4 like in KFC3 performance gains are more at smaller grids. But, for remaining grids, no huge performance gains are realized. Absolute walltime values for V3 and V8 on KFC4 and KFC3 are presented in Table 5-4 and Table 5-5 for comparison. Valgrind

results on KFC4 are presented in Figures 5-8. Cache miss rates appear to be more than the ones for V3 due to a minor decrease in number of data calls as presented in Figures 5-9a and b. This figure compares normalized data calls, D1 misses and L2D misses for V3 and V8 on KFC4. A major reduction in D1 cache misses is seen for grid 200 x 200 explaining walltime gains of up to 10% for this grid.

5.4.1.5 KFC6 Results

Performance results for V8 on KFC6 architectures are presented in Figures 5-10 and 5-11. Figure 5-10 compares walltime values for V3 and V8 on KFC6I and KFC6A. On KFC6I, walltime values show improvements in the range of 12% to 19%, larger gains realized at smaller grids as on KFC3. Relative gains in KFC6 architectures in walltime values are more pronounced than in KFC3 and KFC4.

Table 5-4 Absolute walltime values (in seconds) on KFC4 for V3 and V8

Grid	Walltime-V3	Walltime-V8	% Improvement
30 x 30	11.52	10.45	9.29
60 x 60	49.37	47.79	3.20
80 x 80	99.84	100.66	-0.82
100 x 100	171.87	174.91	-1.77
125 x 125	282.33	287.01	-1.66
150 x 150	413.06	423.86	-2.61
200 x 200	757.27	745.26	1.59
300 x 300	1755.89	1714.92	2.33
400 x 400	3154.64	3192.61	-1.20
500 x 500	4959.73	4993.15	-0.67
600 x 600	7257.02	7148.34	1.50
700 x 700	9973.19	10042.25	-0.69
800 x 800	12852.52	12866.5	-0.11

Table 5-5 Absolute walltime values (in seconds) on KFC3 for V3 and V8

Grid	Walltime-V3	Walltime-V8	% Improvement
30 x 30	12.88	9.71	24.61
60 x 60	57.04	48.55	14.88
80 x 80	113.35	100.87	11.01
100 x 100	201.34	178.41	11.39
125 x 125	332.76	299.63	9.96
150 x 150	476.77	425.66	10.72
200 x 200	846.55	765.73	9.55
300 x 300	1954.73	1753.39	10.30
400 x 400	3429.26	3097.44	9.68

500 x 500	5375.1	6956.37	-29.42
600 x 600	7603.56	7148.34	5.99
700 x 700	9973.19	10042.25	-0.69
800 x 800	12852.52	12866.5	-0.11
900 x 900	16671.13	---	---
1000 x 1000	22035	20158.89	8.51

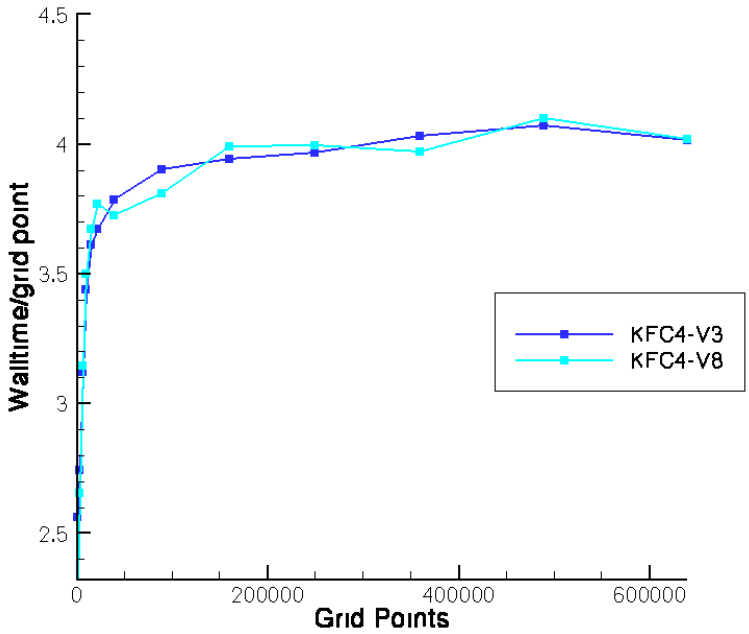
Subroutine *cal_u* in V3:

```
    DO j = 2, njm1
      DO i = 2, nim1
       sumu = sumu +  abs(aewnsp(i,j)%rhs)
      END DO
     END DO
      du = 0.
      CALL tdma_struct (aewnsp, du, 2, nim1, 2, njm1, tdma_iter, tol)
        DO j = 2, njm1
          DO i = 2, nim1
          u (i, j) = u (i, j) + du (i, j)
          upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
        END DO
      END DO
   DO i=1, ni
    j=1
      upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
    j=nj
      upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
   END DO

   DO j=2, nj-1
    i=1
      upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
    i=ni
      upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) / au (i, j)
   END DO
```

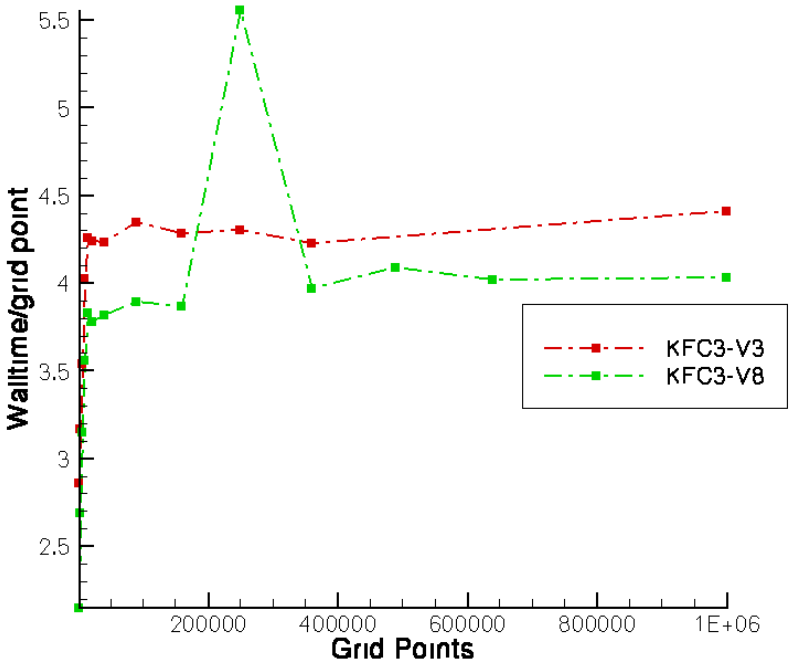
Subroutine *cal_u* in V8:

```
du = 0.
CALL tdma_struct (aewnsp, du, 2, nim1, 2, njm1, tdma_iter, tol)
inv_of_au = 1./au

sumu = 0
 DO j = 2, njm1
  DO i = 2, nim1
   sumu = sumu +  abs(aewnsp(i,j)%rhs)
   u (i, j) = u (i, j) + du (i, j)
   upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) * inv_of_au (i, j)
   END DO
 END DO
  i=1
  DO j=1, nj
  upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) * inv_of_au (i, j)
  END DO
  DO i=2, ni-1
  j=1
    upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) * inv_of_au (i, j)
  j=nj
    upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) * inv_of_au (i, j)
  END DO
    i=ni
  DO j=1, nj
    upp (i, j) = u (i, j) + dpdx (i, j) * vol (i, j) * inv_of_au (i, j)
  END DO
```

Figure 5-6 Implementing Reciprocal Cache and loop merging in subroutine cal_u



(a)



(b)

Figure 5-7 Walltime as a function of grid size for V3 and V8 on (a) KFC4 (b) KFC3

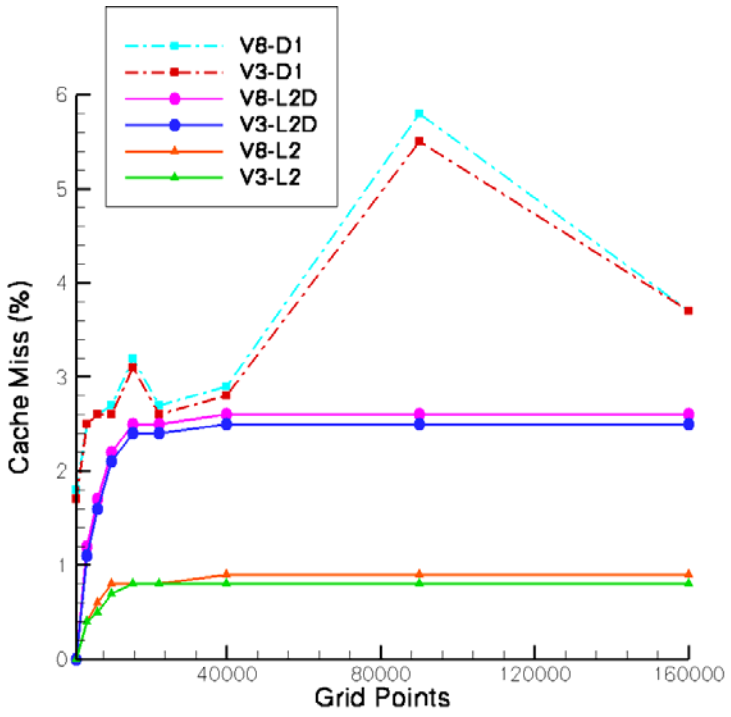
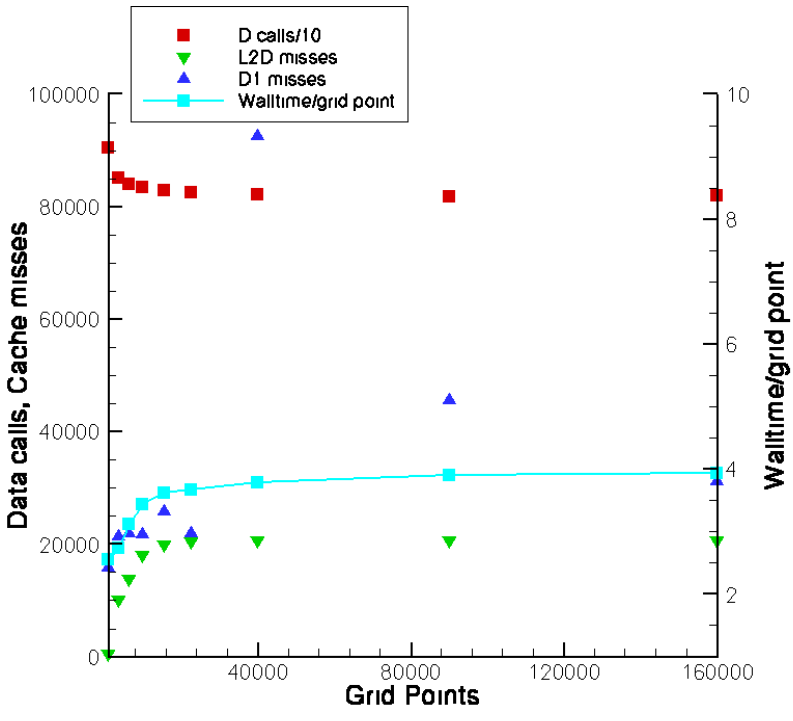
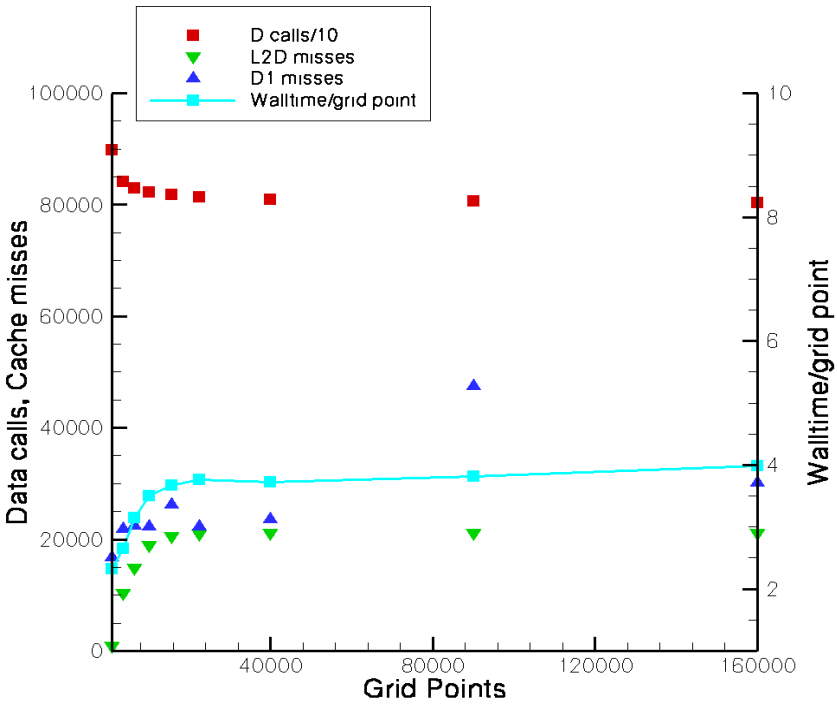


Figure 5-8 Comparison of D1, L2 and L2D cache miss rates for V3 and V8 on KFC4



(a)



(b)

Figure 5-9 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC4 for GHOST (a) V3 (b) V8

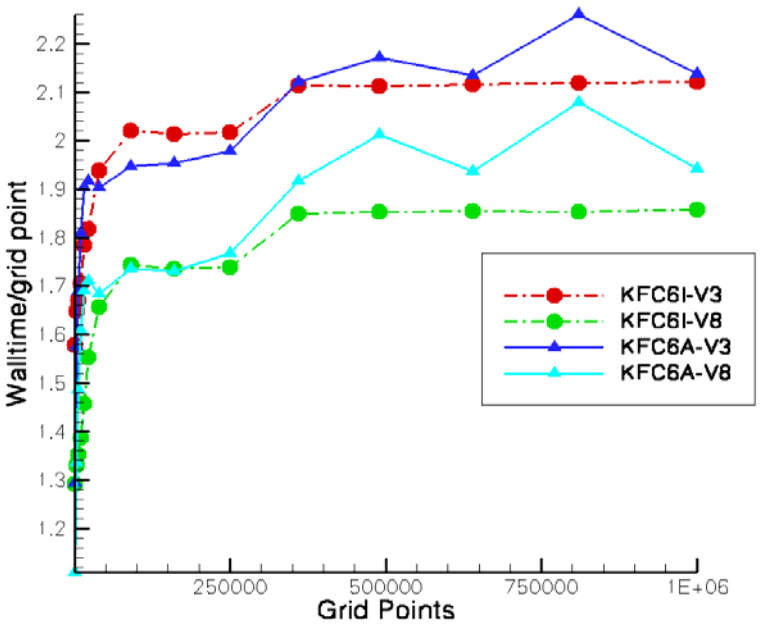
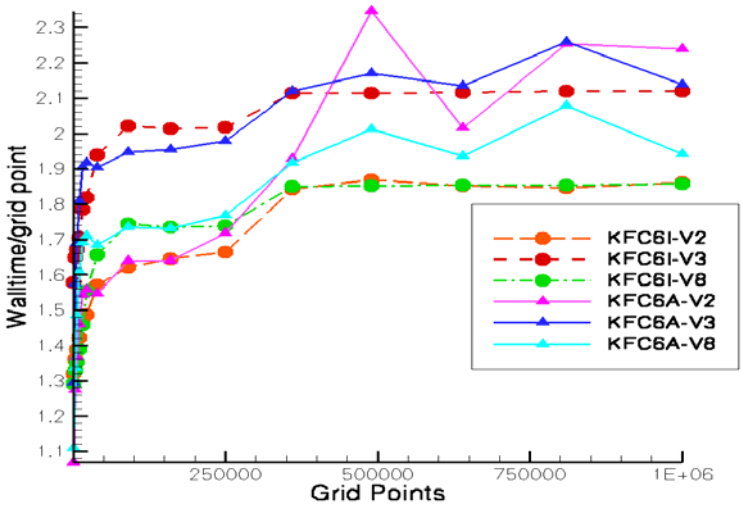
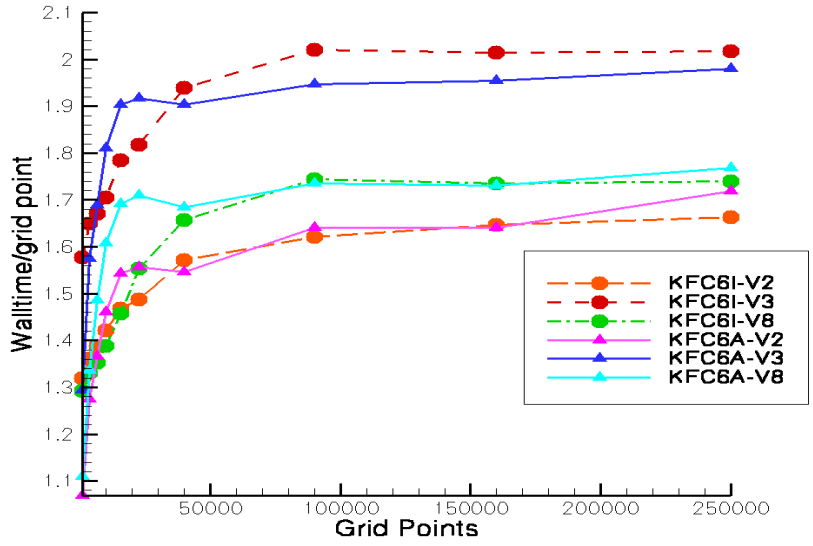


Figure 5-10 Walltime as a function of grid size for V3 and V8 on KFC6I and KFC6A.



(a)



(b)

Figure 5-11 Walltime as a function of grid size for V2, V3 and V8 on KFC6I and KFC6A for all grid sizes (b) zoomed plot for grid points up to 250000

This is not unexpected as V2 was the best tuned code (as presented in chapter 4) on KFC6 architectures and techniques similar to the ones in V2 have been applied inV8.

On KFC6A, performance gains range from 7% to 15%. Similar to KFC6I, gains are more at smaller grids. Figures 5-11a and b present walltime comparisons for V2, V3 and V8 on KFC6 architectures. Also, absolute walltime values for these versions on KFC6 architectures are presented in Tables 5-6a and b.

Table 5-6a Absolute walltime values (in seconds) on KFC6I for V3 and V8

Grid	Walltime-V2	Walltime-V3	Walltime-V8	% Improvement from V2 to V8	% Improvement from V3 to V8
30 x 30	5.94	7.15	5.81	2.14	18.70
60 x 60	24.51	29.6	23.94	2.34	19.13
80 x 80	44.44	53.4	43.29	2.59	18.94
100 x 100	71.11	85.39	69.41	2.39	18.72
125 x 125	114.65	139.44	113.89	0.66	18.32
150 x 150	167.38	204.51	174.77	-4.41	14.54
200 x 200	314.19	385.99	331.45	-5.49	14.13
300 x 300	729.54	909.99	785.18	-7.63	13.72
400 x 400	1317.01	1613.48	1388.41	-5.42	13.95
500 x 500	2079.46	2517.36	2174.18	-4.56	13.63
600 x 600	3315.71	3804.55	3330.30	-0.44	12.47
700 x 700	4578.25	5178.44	4538.26	0.87	12.36
800 x 800	5926.16	6769.47	5932.76	-0.11	12.36
900 x 900	7477.15	8599.42	7506.82	-0.40	12.71
1000 x1000	9308.88	10588.74	9292.79	0.17	12.24

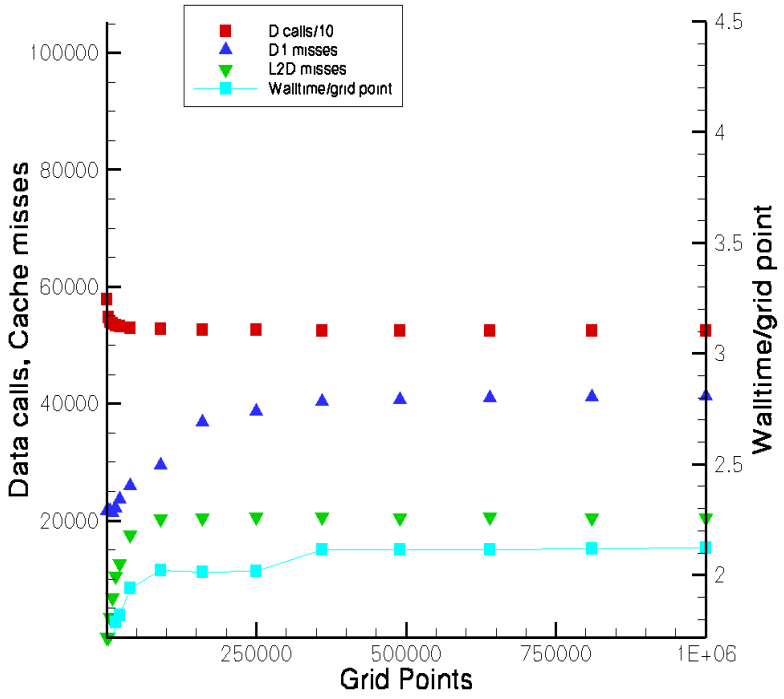
Table 5-6b Absolute walltime values (in seconds) on KFC6A for V3 and V8

Grid	V2-walltime(s)	V3-walltime (s)	V8-walltime(s)	% Improvement from V2 to V8	% Improvement from V3 to V8
30 x 30	4.81	5.82	4.99	-3.79	14.22
60 x 60	22.95	28.35	24.05	-4.77	15.18
80 x 80	43.73	54.06	47.55	-8.73	12.05
100 x 100	73.01	90.5	80.39	-10.10	11.18
125 x 125	120.6	148.73	132.14	-9.56	11.16
150 x 150	175.19	215.6	192.37	-9.81	10.77
200 x 200	309.22	380.74	336.81	-8.92	11.54
300 x 300	738.11	876.41	780.90	-5.80	10.90
400 x 400	1312.16	1562.98	1385.17	-5.56	11.38
500 x 500	2148.26	2474.13	2210.29	-2.89	10.66
600 x 600	3470.18	3817.54	3449.56	0.59	9.64
700 x 700	5749.77	5318.88	4929.94	14.26	7.31
800 x 800	6448.75	6830.06	6197.23	3.90	9.27
900 x 900	9126.26	9153.32	8422.71	7.71	7.98
1000x1000	11191.61	10690.38	9707.16	13.26	9.20

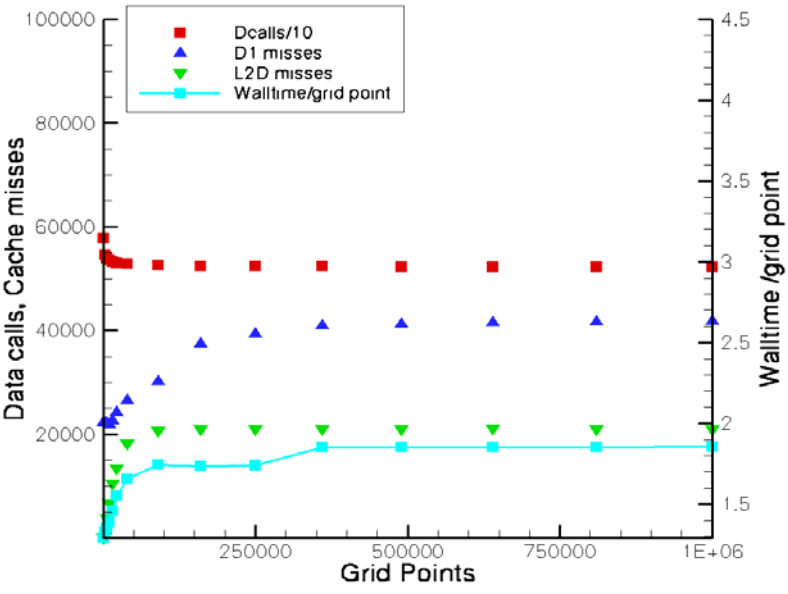
- **V3 vs. V8 on KFC6I and KFC6A:** V8 performs better than V3 for all grid sizes tested on KFC6I and KFC6A. This is not unexpected because performance optimization techniques that were implemented in V8 were similar to the ones in V2 (and V2 was the best code for KFC6I as discussed in chapter 4).
- **V2 vs. V8 on KFC6I:** As seen from Table 5-6a, for small grids and large grids V8 is equal or better than V2 on KFC6I. For moderate grids, V2 is better
- **V2 vs. V8 on KFC6A:** In V8, performance gains of up to 14% are realized at larger grids on KFC6A. However, V2 tends to perform better than V8 at smaller grids. Valgrind data was not collected on KFC6A and so no proper explanation is available.

As discussed at the beginning of this chapter, the focus in V8 was to extend the optimization techniques that were applied to subroutine *cont* to other subroutines (*cal_u* and *cal_v*) to test if gains realized in V2 can be realized in V8 as well. As presented in Figures 5-11a and b and Tables 5-7a and b, on KFC6I performance gains in V8 are realized only at larger grids (beyond 600 x 600). V8 performs better than V2 by a minor margin (2-3%) on KFC6I while it performs better (up to 13%) than V8 on KFC6A for larger grids. Figures 5-12 a and b present normalized values of data calls, L2D misses and D1 misses on KFC6I for V3 and V8 for comparison. As seen in the Figure 5-12 for V8, there is a drop in number of data calls and L2D misses at larger grids (beyond 500 x 500) although this does not translate to improvements in walltime on KFC6I as can be seen from Table 5-6a.

From the analysis below, it is clear that using data structures (in V3) proved to be beneficial for larger grids while it is more detrimental for moderate grids and yields mixed results for smaller grids. The negative effect of data structures on walltime is more pronounced on KFC6 architectures.



(a)



(b)

Figure 5-12 Comparisons of normalized walltime and normalized number of data calls (divided by 10), D1 cache misses and L2D cache misses on KFC6I for GHOST (a) Version 3 (b) Version 8

5.5 AIR FOIL TEST CASE

In order to test the universality of the presented tuning techniques, the flow over a NACA 4318 airfoil inside a 24 x 24 inch wind tunnel section was chosen as a second test case. The experimental setup for this case is as shown in Figure 5-13. The computational grid for the airfoil comprises of 1031 x 120 grid points. The overall computational grid consisting of two-dimensional multi-zonal blocks is shown in Figure 5-14. The airfoil grid overlaps the central background grid. This background grid is surrounded by eight other rectangular grids.

On the outer boundary, the left (inlet) boundary is fixed with a uniform dimensionless inlet velocity $u_\infty = 1.0$ and the upper and lower boundary condition are no-slip wall boundaries representing the top and bottom of the wind tunnel test section. For the airfoil blocks, the inner boundary condition is a no-slip wall boundary condition, and the outside boundary is set to "overlap" which allows the background grid points to be overlapped by the airfoil block grid points to interpolate values from the foreground airfoil grid points.

Computation information between adjacent blocks is exchanged by two ghost points. All the parameters chosen in the computation are dimensionless. Information regarding the grid sizes of the individual blocks in the grid is presented in Table 5-7.

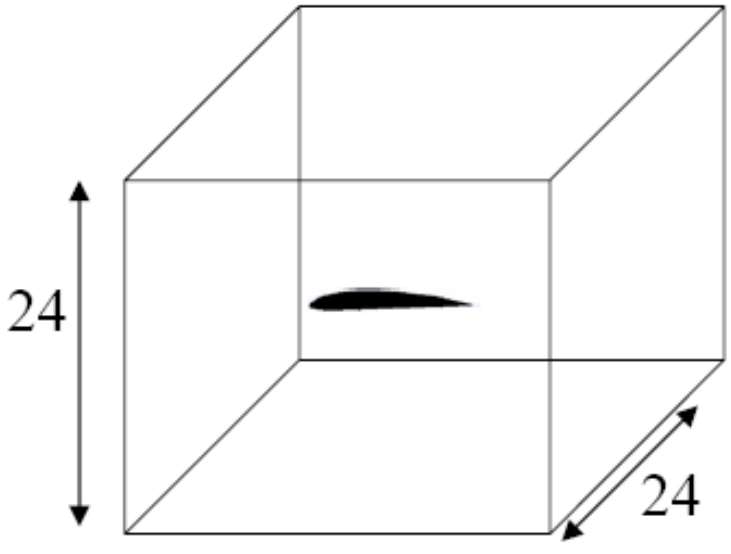


Figure 5-13 24 x 24 inch experimental test section

Table 5-7 Block sizes of individual zones on airfoil grid

Zone	Grid Size	Zone	Grid Size
	i x j		*i x j*
1	1031 x 120	6	300 x 40
2	50 x 100	7	50 x 40
3	300 x 100	8	50 x 40
4	50 x 100	9	300 x 40
5	50 x 40	10	50 x 40

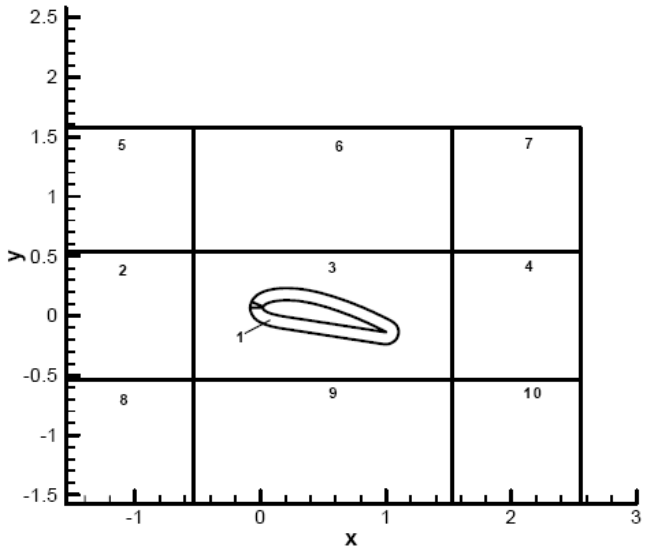


Figure 5-14 Grid used for 24 x 24 inch wind tunnel section

5.5.1 PERFORMANCE TEST RESULTS

Table 5-7 contains the information regarding the grid sizes of the individual blocks in the grid. Walltime values for the above discussed test case have been recorded on KFC3 and KFC6I. Although the airfoil grid block (zone 1) contains relatively more number of points than the surrounding blocks, no external/internal blocking has been applied to this block in order to test the optimization techniques presented in chapter 4. The code was run on a single node for 5000 iterations. A Reynolds number of 25000 was used. A comparison of the absolute walltime values for 5000 iterations is presented in the Table 5-8.

Table 5-8 Comparison of walltime values for various versions of GHOST for flow over a NACA 4318 airfoil on various hardware platforms

Platform	V0	V1 (%Improvement compared to V0)	V2 (%Improvement compared to V1)	V3 (%Improvement Compared to V1)	V8 (% Improvement compared to fastest version so far)
KFC3	6757.87	5125.51 (24.15%)	4683.14 (8.63%)	5042.54 (1.61%)	4590.09 (1.98% - fastest)
KFC6I	2899.3	2288.25 (21.07%)	1916.56 (16.24% - fastest)	2298.37 (- 0.44%)	1959.03 (- 2.21%)

On KFC3, V8 performs better than any other version of GHOST while on KFC6, V2 performs better than any other version. This is not unexpected from the results discussed at the beginning of this chapter and in chapter 4. V3 has in it arrays of data structures replaced by arrays and this technique appears to be detrimental for the performance of the code for this test case on both KFC3 and KFC6. However, performance tuning techniques have largely been beneficial for airfoil test as well, thus proving that the code optimization techniques do result in better performance for other test cases as well.

5.6 SUMMARY

Results of external and internal blocking techniques were briefly presented at the beginning of this chapter. While the performance gains attained from application of external and internal blocking techniques were impressive, each of these techniques has its own disadvantages. Implementing external blocking is more time consuming on complicated grids while internal blocking has problems in terms of accuracy of the results for complicated grids. This represented an opportunity to carry out further tuning effort that in addition to the efforts discussed in chapter 4 with a goal to realize further gains and to circumvent the problems of external and internal blocking techniques. Results of further optimization effort on GHOST have been presented with gains up to 25% for smaller grids and up to 9% on larger grids (1 million grid points) when

compared to V3 on KFC3 and KFC4. While these gains were sporadic, performance gains up to 20% were realized for all grid sizes tested on KFC6I and KFC6A. Performance gains in V8 when compared to V2 are meager. Only about 2% gains were realized for smaller grids in V8 when compared to V2 on KFC6I. For other grids, performance is almost the same for both the codes. On KFC6A in V8, performance gains up to 14% were realized. These gains were only for larger grids. To conclude V8 is better than V2 only for larger grids on KFC6 architectures while gains were more pronounced at smaller grids on KFC3 and KFC4. Later, performance tests were conducted on KFC3 and KFC6I on a NACA 4318 airfoil to establish that the code optimization techniques do result in better performance for other test cases as well.

CHAPTER-6

6. CONCLUSIONS AND FUTURE WORK

6.1 SUMMARY AND CONCLUSIONS

In the present work, results of optimization effort on a 2D structured CFD code GHOST are presented. The beginning part of the work consists of a discussion of why faster processors and newer computers do not necessarily translate into better performance of scientific codes. Later, background information about memory architecture is presented. A description of the commodity clusters on which tuning effort has been carried out is then presented. Later, various code optimization techniques were presented. Parts of the GHOST code that were detrimental to its performance were identified and the tuning effort was carried out in stages.

Optimization effort primarily began with identifying the bottlenecks in the original version of GHOST (V0) and creating a baseline in terms of performance. Initial tests were conducted using the laminar lid-driven cavity flow test case. To measure the performance of the code, cache behavior was analyzed with the Valgrind toolkit while walltime information was captured using UNIX functions. Walltime values for 5000 iterations were normalized by the number of grid points of the computational problem and by the number of iterations so as to eliminate the effect of increasing grid size. If the code had been running efficiently on a single node, the normalized walltime sould have increased as we moved from grids that fit into cache to grids that are considerably larger than cache. After that, the normalized walltime would have essentially remained same even as the size of grid increased. However, this was not what was observed. The performance of the code was sub-optimal on a single node and was characterized by increasing normalized walltime values with increasing grid sizes. The performance was plagued by high cache miss rates due to mismatch between data access and data storage in memory along with redundant mathematical activity in the code leading to unnecessary data calls. Sub-optimal performance of this code on a single node was traced to heavy cache misses.

Sub-optimal performances of GHOST on a single node lead to its super linear behavior across multiple nodes because speedup is calculated based on the performance

of the code on a single node. The fact that this super linear behavior was observed on all machines tested despite the differences in the year of construction (KFC3 in 2003 through KFC6 in 2006) and disparity in hardware (for example KFC6A has an AMD processor while KFC6I has an Intel processor) and networks (relatively fast on KFC3, relatively slow on KFC4) made a strong case for the need for tuning GHOST

Optimization efforts on GHOST can largely be classified into two parts and have proved largely successful. The first part of the tuning effort (carried out till December 2004) primarily consisted of optimization effort on KFC3 and KFC4. Major performance problems that were identified in the original version of GHOST were addressed in this part. This part of the tuning effort essentially has 3 stages to it. In each stage, various code optimization techniques were implemented and the performance of the code was measured in terms of walltime values and cache behavior (L2D misses, D1 misses, Data calls, L2D cache miss rate, and L2 cache miss rate). These values were compared with the values in previous versions of the code.

In the first stage, the order of i, j sweeps was corrected to the cache conserving form so that there is no mismatch between the order of data access and data storage. This yielded 5% (on smaller grids) to 85% gains (on larger grids) on KFC3 and KFC4 while performance gains were up to 50% on KFC6I while on KFC6A, they were 28%. Lower gains are because of the faster processor and bigger caches (when compared to KFC3 and KFC4) along with faster Front Side Bus (FSB). Walltime improvements were attributed to improvements in D1 cache behavior. On KFC6I, D1 misses were almost 50% lesser for V1 when compared to V0.

The second stage of tuning effort was focused on reducing redundant mathematical operations with a goal of reducing floating point operations. In this stage, the focus was on subroutine *cont* as it was the most expensive. Repeated divisions inside loops were replaced by pre calculated reciprocal values thereby reducing the frequency of division as division operations have higher latency when compared to other mathematical operations. When nested loops were next to each other, proper analysis was done if there is a possibility of merging them. If it turned out that the merge of nested loops did not alter the algorithm of the code, loop merging (a technique discussed in chapter 2) was done to achieve *temporal locality* as the number of iterations that separate successive

accesses to a given reused data is reduced if same data elements were being referenced in two consecutive loops before their merge. Conditional statements inside nested loops were also removed by re-writing the code without conditional checks. In V2, performance gains on KFC3 and KFC4 were not huge as the focus was only on tuning one subroutine (*cont*). On the other hand, improvements up to 10% (when compared to V1) were noticed on KFC6 architectures due to decrease in number of data calls and D1 misses.

V3 is the result of the third stage of the tuning effort. In order to understand the effect of usage of data structures, the changes done to the code from V1 to V2 have not been implemented in V3. Thus, V3 is V1 with data structures implemented in place of arrays to aid in data fetch. The focus of this tuning stage was to avoid data misses that might be possible due to usage of arrays in the code as explained in chapter 2. On KFC3 and KFC4, performance gains up to 20% were noticed for larger grids (1 Million grid points). However, there was performance degradation in V3 (when compared to V1 and V2) on KFC6 architectures. Walltime behavior in this case could not simply be explained by cache miss data. The results of efforts of the above presented optimization techniques were incorporated into LeBeau *et. al.* [52]. This paper included further optimization efforts on GHOST beyond the scope of this thesis, but based on the work presented so far. Results of their performance tuning effort were summarized at the end of chapter 4.

From performance tuning efforts in stage 1, the best optimized version of the code was within a factor of 2 of the estimated optimal performance over all the tested grid sizes and the overall performance improvement for this case relative to the original code ranged from 20% faster for small grids to over 6 times faster for the largest.

The second part of the tuning effort comprises of improvising on the performance effort carried out on GHOST between December 2004 and November 2008. Two techniques that were analyzed were External blocking and Internal blocking. Results of the tuning efforts by Palki [53] for a laminar lid-driven cavity flow test case were presented in chapter 5. With the application of External blocking technique, performance improvements up to 28% were observed on KFC4 for all grid sizes tested (up to 360000 grid points) achieved on previously tuned (V3) laminar version of GHOST. With Internal

blocking, improvements up to 26% were observed on KFC4. Improvements attained by using both these techniques were less on KFC6 architectures due to advanced hardware.

While the performance gains attained from application of external and internal blocking techniques were impressive, each of these techniques has its own disadvantages. Implementing external blocking is more time consuming on complicated grids while internal blocking technique has problems in terms of accuracy for complicated grids. This represented an opportunity to carry out further tuning effort that was discussed in chapter 4 so as to realize further gains and to circumvent the problems of external and internal blocking techniques. Results of further optimization effort (similar techniques applied in V2) on GHOST have been presented with gains up to 25% for smaller grids and up to 9% on larger grids (1 million grid points) when compared to V3 on KFC3 and KFC4. While these gains were sporadic, performance gains up to 20% were realized for all grid sizes tested on KFC6I and KFC6A. Later, performance tests were conducted on KFC3 and KFC6I on a NACA 4318 airfoil to establish that the code optimization techniques do result in better performance for other test cases as well.

From the results of optimization effort carried out in stages 1 and 2, it can be concluded the best optimized version of GHOST on KFC3 and KFC4 was V8 while it was V2 and V8 performed equally well on KFC6 architectures. Arrays of data structures implemented in place of arrays in V3 resulted in performance gains on KFC3 and KFC4 while they were detrimental on KFC6I and KFC6A. However, the techniques implemented in V8 resulted in more gains on KFC6 architectures almost nullifying the loss due to implementation of data structures in V3.

6.2 IMPACT OF CURRENT WORK

The impact of the current work is realized in multiple projects because optimized versions of the GHOST code based on the presented techniques are being successfully used in them. Optimized versions of GHOST have saved hundreds of hours of CPU time and the projects have been completed within a fraction of the time it would have taken if the original version of GHOST had been used for CFD simulations. Some of these projects use an airfoil similar to the one presented in chapter 5.

For example, in studies on “Applying Genetic Algorithms (GA) to Complex Fluid Dynamics Simulations” [73], “Application of Genetic Algorithms and Neural Networks

to Unsteady Flow Control Optimization" [74] optimized versions of GHOST code based on techniques presented in this work were used to perform CFD simulations. The test problems were similar in these studies except that the first one was a steady flow on a NACA 0012 airfoil at an 18 degree angle of attack and a Reynolds number of 500,000 while the latter one was an unsteady flow on the same airfoil with same values of the angle of attack and Reynolds number. Commodity cluster architectures KFC5, KFC6 and KFC6A that were discussed in this work were used as computational platforms. In the former study, 2800 simulations (50 generations) were performed using GHOST and each generation took 29 hours and 23 hours respectively on 19 nodes on KFC5 and KFC6A.

In a different study "Experimental and Computational Investigation of a Modified NACA 4415 in Low-Re Flows", [75] the test case was flow over a NACA 4415 airfoil with Reynolds numbers ranging from 2.5×10^4 to 10×10^4 and over a range of angles of attack. The computational grid comprised of 85000 grid points. With a baseline dimensionless timestep of t=0.0001, 10 subiterations were run on three processors on KFC4. The runtime was 12 hours per dimensionless time unit for laminar simulations and 21 hours for transitional simulations. Optimized versions of GHOST code based on techniques presented in this work were used to perform CFD simulations in this work as well. These are few examples of many instances in which optimized versions of the GHOST code have been used.

As demonstrated in chapter 5, for a steady, laminar flow on a NACA airfoil 4318 performance gains up to 32% were realized on KFC3. Similar or better gains have been achieved when the techniques presented were applied to the above presented test cases. For turbulent flows, performance gains were up to 50% effectively cutting down the simulation times into half. Thus, the optimized versions of the GHOST code had a major impact on projects undertaken by the UK Cluster Fluid Dynamics Group at the University of Kentucky.

6.3 FUTURE WORK

The present work has been largely successful in improving the performance of GHOST on commodity cluster architectures. It has been quite successful in attaining its main objective *viz.* to reduce cache miss rates to the lowest possible number. The relation

between the performance of a code and its cache behavior has been studied in detail. Unexpected behaviors of the code with implementing data structures on modern (KFC6) architectures need to be studied in detail as such behavior could not be traced to cache behavior of the code. Also, based on walltime recorded for V8 on KFC6 architectures, initial conclusion was V8 without data structures would be the optimum code on KFC6 architectures. Accordingly, a new code V9 was constructed from V8 re-implementing arrays instead of arrays of data structures. However, walltime values for V9 do not prove that V9 is a better code than V8 on KFC6 architectures. As the effects of using data structures in GHOST is not clear on dual-core KFC6 machines, further analysis is required. A probable starting point is to explore this relation of data structures on walltime values by experimenting with a new data structure using a different set of variables.

REFERENCES

1. www.autofieldguide.com
2. http://www.intel.com/business/casestudies/BMW_Sauber_F1_case_study.pdf
3. www.cfdreview.com
4. www.boeing.com
5. Baggett, J. S. 1997 -- Some modeling requirements for wall models in large eddy simulation. *Annual Research Briefs 1997*, Center for Turbulence Research, NASA Ames/Stanford Univ., 123-134.
6. http://www.aps.org/units/dfd/meetings/upload/Spalart_DFD04.pdf
7. Paul r. Woodward, David h. Porter, Igor sytine, S. E. Anderson, Arthur a. Mirin, B. C. Curtis, Ronald h. Cohen, William P. Dannevik, Andris M. Dimits, Donald E. Eliason Karl-Heinz Winkler, Stephen W. Hodson -- Very high resolution simulations of compressible, turbulence on IBM-SP system, in Proceedings of the ACM/IEEE SC99 Conference, Portland, Oregan, November 13-18 1999, http://www.supercomp.org/sc99/proceedings/index.htm
8. W.K. Anderson, W.D. Gropp, D.K. Kaushik, D.E. Keyes, B.F. Smith – Achieving high sustained performance in an unstructured mesh CFD application, in Proceedings of the ACM/IEEE SC99 Conference, Portland Oregan, November 13-18 1999, http://www.supercomp.org/sc99/proceedings/index.htm
9. Donald J. Becker, Thomas Sterling, Daniel Savarese, John E. Dorband, Udaya A. Ranawak, and Charles V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings, International Conference on Parallel Processing*, 1995.
10. T. E. Anderson, D. E. Culler, and D. A. Patterson. A case for networks of workstations: NOW. *IEEE Micro*, Feb. 1995.
11. Distributed Terascale Facility to Commence with $53 Million NSF Award. http://www.nsf.gov/od/lpa/news/press/01/pr0167.htm.
12. T. Hauser, T. I. Mattox, R. P. LeBeau, H. G. Dietz and P. G. Huang, "High-cost CFD on a low-cost cluster", Supercomputing 2000, November 2000.
13. http://www.myricom.com/myrinet/overview/
14. http://www.hpcwire.com – IBM Roadrunner takes the Gold in the Petaflop Race – by Michael Feldman – June 09, 2008.
15. Thomas Sterling and Ian Foster, "Proceedings of the Petaflops Systems Workshops," Technical Report CACR-133, California Institute of Technology, Oct. 1996.
16. P. Moin and K. Kim, "Tackling turbulence with supercomputers", Scientific American, January 1997, vol. 276, No 1, pp 62-68
17. Markus Nordén, Malik Silva, Sverker Holmgren, Michael Thuné, Richard Wait – Implementation issues for High Performance CFD.

18. T. Mowry, M. Lam and A. Gupta – Design and evaluation of a compiler algorithm for prefetching. In Proceedings of the *Fifth International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62-73, Boston, MA, October 1992.
19. K. Beyls, "Software Methods to Improve Data Locality and Cache Behavior," - Doctoraatsproefschrift Faculteit Toegepaste Wetenschappen, Universiteit Gent, 2004.
20. D. A. Patterson and J. L. Hennesey, Computer Organization and Design, Morgan Kaufmann Publishers, San Francisco, 1998.
21. http://en.wikipedia.org/wiki/Distributed_memory
22. www.pcguide.com
23. http://en.wikipedia.org/wiki/IBM_PC
24. www.amazon.com
25. Fotheringham, J. 1961. "Dynamic storage allocation in the Atlas computer, including an automatic use of a backing store." ACM Communications 4, 10 (October), 435-436.
26. Kilburn, T., D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner. 1962. "One-level storage system." IRE Transactions EC-11, 2 (April), 223-235.
27. P. Mazzucco, "The Fundamentals of Cache", Systemlogic.net, October 2000.
28. Wilkes, M.V- Slave Memories and Dynamic Storage Allocation. IEEE Trans. EC-14, 1965, pp 270-271.
29. Roger W. Hockney, C.R. Jesshope – Parallel Computers 2: Architecture, Programming, and Algorithms -- Published by CRC Press, 1988 ISBN 0852748116, 9780852748114 -- 625 pages.
30. J. Tyson – "How Computer Memory Works", HowStuffWorks – www.howstuffworks.com
31. www.hardwaresecrets.com
32. http://en.wikipedia.org/wiki/Cache
33. http://www.webopedia.com/TERM/S/SRAM.html
34. J.L. Hennessy and D.A. Patterson, Computer Architecture – A Quantitative Approach, Morgan Kaufmann Publishers, third edition, 2002.
35. Douglas W. Clark – Cache Performance of the VAX-11/780. ACM Transactions on Computer systems, 1(1): 24-37, 1983.
36. D. Fenwick, D. Foley, W. Gist, S. VanDoren and D. Wissell – The AlphaServer 8000 Series: High end server platform development. Digital Technical Journal, 7(1): 43-65, 1995
37. http://en.wikipedia.org/wiki/CPU_cache

38. J. L. Baer, “2k papers on caches by Y2K: Do we need more?” Keynote address at the 6th International Symposium on High-Performance Computer Architecture, January 2000.

39. M. Jahed Djomehri , Rupak Biswas - Performance Enhancement Strategies for Multi-Block Overset Grid CFD Applications -- Volume 29 , Issue 11-12 (November/December 2003) Special issue: Parallel and distributed scientific and engineering computing - Pages: 1791 - 1810 Year of Publication: 2003 - ISSN:0167-8191.

40. Stefan Goedecker, Adolfy Hoisie -- Performance optimization – Numerically Intensive Codes – Publisher: The Society for Industrial and Applied Mathematics; ISBN 0-89871-484-2.

41. K. S. McKinley, S. Carr and C. –W. Tseng, “Improving data locality with loop transformations”, ACM Transactions on Programming Languages and Systems, 18(4):424-453, July 1996.

42. Michael J. Flynn, Stuart F. Oberman -- Design Issues in Floating-Point Division – Technical Report: CSL-TR-94-647; Computer Architecture and Arithmetic Group, Stanford University.

43. Stuart F. Oberman, Michael J. Flynn -- An analysis of division algorithms and implementations – Technical Report: CSL-TR-95-675; Computer Architecture and Arithmetic Group, Stanford University.

44. Craig C. Douglas, Jonathan Hu, Markus Kowarschik, Ulrich Rude, Christian Weiss – Cache Optimization For Structured and Unstructured Grid Multigrid – Electronic Transactions on Numerical Analysis Journal – volume 10, 2000 – pp 21-40, ISSN: 1068-9613.

45. Dinesh K. Kaushik, David E. Keyes, and Barry F. Smith -- On the Interaction of Architecture and Algorithm in the Domain-based Parallelization of an Unstructured-grid Incompressible Flow Code -- Proceedings of the 10th Intl. Conf. on Domain Decomposition Methods, pages 311-319, 1998.

46. http://fun3d.larc.nasa.gov/

47. Thomas Hauser, Timothy I. Mattox, Raymond P. LeBeau, Henry G. Dietz, P. George Huang, "High-Cost CFD on a Low-Cost Cluster," sc,pp.55, ACM/IEEE SC 2000 Conference (SC 2000), 2000.

48. S. Kadambi and J.C. Harden, “Accelerating CFD applications by improving cached data reuse”, ssst, p. 120, 27th Southeastern Symposium on System Theory (SSST’95), 1995

49. W. Gropp, D. Kaushik, D. Keyes, and B. Smith, “Performance modeling and tuning of an unstructured mesh CFD application”, Proceedings of SC2000.

50. W. Gropp, D. Kaushik, D. Keyes, and B. Smith, “High performance parallel implicit CFD”, Parallel Computing, 27(4):337—362, March 2001.

51. S. Gupta, “Performance evaluation and optimization of the unstructured CFD code UNCLE”, Thesis, University of Kentucky, May 2006.

52. R. P. LeBeau, P. Kristipati, S. Gupta, H. Chen, P. G. Huang, "Joint Performance Evaluation and Optimization of Two CFD Codes on Commodity Clusters", AIAA – 2005 – 1380, January 2005.

53. Anand Palki – "Cache optimization and performance evaluation of a structured CFD code – GHOST", Thesis, University of Kentucky, December 2006.

54. Y. B. Suzen and P. G. Huang, "Numerical simulations of wake passing on turbine cascades", AIAA-2003-1256, 2003.

55. Y. B. Suzen and P. G. Huang, "Predictions of separated and transistional boundary layers under low-pressure turbine airfoil conditions using an intermittency transport equation", Journal of Turbomachinery, Vol. 125, No.3, Jul. 2003, pp. 455-464.

56. Katam, V., LeBeau , R.P., and Jacob, J.D. "Simulation of Separation Control on a Morphing Wing with Conformal Camber", AIAA-2005-4880, 2005.

57. L. Huang, P. G. Huang, R. P. LeBeau and Th. Hauser, "Numerical study of blowing and suction control mechanism on NACA0012 airfoil", Journal of Aircraft, Vol. 41, 2004, pp. 1005 – 1013.

58. L. Huang, P. G. Huang, R. P. LeBeau and Th. Hauser, "Optimization of blowing and suction control on NACA 0012 airfoil using a genetic algorithm", AIAA -2004-0423, 2004.

59. Suzen, Y. and Huang, G., "Simulations of Flow Separation Control using Plasma Acuators", AIAA-2006-877, 2006.

60. C. M. Rhie and W. L. Chow, "Numerical study of the turbulent flow past an airfoil with tailing edge separation", AIAA Journal, Vol. 21, 1983, pp. 1523-1532.

61. Valgrind Home – http://www.valgrind.org

62. http://kcachegrind.sourceforge.net

63. http://www.cfg-online.com/Wiki/Overset_grids

64. F. N. Felten and T. S. Lund, "Kinetic energy conservation issues associated with the collocated mesh scheme for incompressible flow", Journal of Computational Physics, November 7, 2005.

65. Jeff Layton, "Tips and tricks for tuning CFD codes", White Paper for Linux Networx.

66. U. Ghia, K. N. Ghia and C. T. Shin, "High-resolution for incompressible flow using the Navier-Stokes equations and a multigrid method", Journal of Computational Physics, Vol. 48, 1982, pp. 387 – 411.

67. http://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-_TDMA_(Thomas_algorithm)

68. Spinnato, P., Van Albada, G.D., and Sloot, P.M.A., "Performance Analysis of Parallel N-body Codes", *Proceedings of the sixth annual conference of the Advanced School of Computing and Imaging*, ASCI Delft, Jun. 2000, pp. 213-220.

69. Ewing, A. and Henty, D., "Edinburgh Parallel Computing Centre: T3D Technical Report", EPCC-TR95-05, Version 1.0, Oct. 1995.

70. Agarwal, G., and Saltz, J., "Interprocedural Compilation of Irregular Applications for Distributed Memory Machines", *Proceedings of IEEE/ACM SC1995*, 1995.

71. Nielsen, E.J., Anderson, W.K., and Kaushik, D.K., "Implementation of a Parallel Framework for Aerodynamic Design Optimization on Unstructured Meshes", *Proceedings of Parallel Computational Fluid Dynamics '99*, May, 1999

72. URL: http://warewulf-cluster.org/.

73. Raymond P. LeBeau, Thomas Hauser, Narendra Beliganur, Daniel G. Schauerhamer, "Applying Genetic Algorithms to Complex Computational Fluid Dynamics Simulations", 45th AIAA Aerospace Sciences Meeting and Exhibit, 8-11 January 2007, Reno, Nevada.

74. Narendra K. Beliganur, Raymond P. LeBeau, Thomas Hauser, "Application of Genetic Algorithms and Neural Networks to Unsteady Flow Control Optimization", 18th AIAA Computational Fluid Dynamics Conference, 25 – 28 June 2007, Miami, FL.

75. Vamsidhar Katam, Raymond P. LeBeau, Jamey D. Jacob, "Experimental and Computational Investigation of a Modified NACA 4415 in Low-Re-Flows", AIAA-2004-4972 22nd Applied Aerodynamics Conference and Exhibit, Providence, Rhode Island, Aug. 16-19, 2004.

76. Rajesh Bhaskaran, Lance Collins "Introduction to CFD Basics"

VITA

Pavan K. Kristipati was born on April 5th 1981 in Anantapur, India. He received Bachelors degree in Mechanical Engineering from Jawaharlal Nehru Technological University, Hyderabad, India in June 2002. Currently, he is working on completing this Masters Program while working fulltime.

Professional Summary:

1) Programmer/Analyst – Current
Marlabs, Inc.
Edison, NJ.
2) Teaching / Research Assistant – Spring 2003 – Fall 2004
Department of Mechanical Engineering,
University of Kentucky, Lexington, KY.

Scholastic Honors:

Kentucky Graduate Scholarship, Fall 2002 – Fall 2004
University of Kentucky, Lexington, KY.

Papers and Conferences:

1. Raymond P. LeBeau, Jr., Pavan Kristipati, Saurabh Gupta, Hua Chen, George P. G. Huang – "*Joint Performance Evaluation and Optimization of Two CFD Codes On Commodity Clusters*" – 43rd Aerospace Sciences Meeting and Exhibit – January 10-13, 2005, Reno, Nevada.
2. Gupta S., LeBeau Jr. R.P., Kristipati P., Huang P.G., (2005) *"Performance Optimization of a Structured and Unstructured Code on Commodity Clusters,"* presented at 31st AIAA Dayton- Cincinnati Aerospace Sciences Symposium, Dayton, Ohio. March, 2005.